Functional Arabic Morphology

Formal System and Implementation

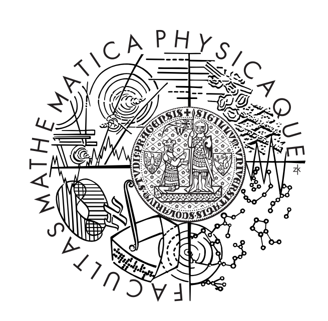
اَلصَّرفُ العَرَبِيُّ الوَظِيفِيُّ

اَلنِّظَامُ الرَّسمِيُّ وَتَحقِيقُهُ

Otakar Smrž

Doctoral Thesis

Prague 2007



INSTITUTE OF FORMAL AND APPLIED LINGUISTICS
FACULTY OF MATHEMATICS AND PHYSICS
CHARLES UNIVERSITY IN PRAGUE

Supervisor

Mgr. Barbora Vidová Hladká, Ph.D.
Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague

Opponents

Doc. RNDr. Jan Hajič, Dr.
Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague

Nizar Y. Habash, Ph.D.
Center for Computational Learning Systems
Columbia University



to my family and friends

Abstract

Functional Arabic Morphology is a formulation of the Arabic inflectional system seeking the working interface between morphology and syntax. ElixirFM is its high-level implementation that reuses and extends the Functional Morphology library for Haskell.

Inflection and derivation are modeled in terms of paradigms, grammatical categories, lexemes and word classes. The computation of analysis or generation is conceptually distinguished from the general-purpose linguistic model.

The lexicon of ElixirFM is designed with respect to abstraction, yet is no more complicated than printed dictionaries. It is derived from the open-source Buckwalter lexicon and is enhanced with information sourcing from the syntactic annotations of the Prague Arabic Dependency Treebank.

MorphoTrees is the idea of building effective and intuitive hierarchies over the information provided by computational morphological systems. MorphoTrees are implemented for Arabic as an extension to the TrEd annotation environment based on Perl.

Encode Arabic libraries for Haskell and Perl serve for processing the non-trivial and multi-purpose ArabTEX notation that encodes Arabic orthographies and phonetic transcriptions in parallel.

In the dark, I will tell you I know
myself three times the way I do . . .

Lucie, the nightly *Noc*

Preface

There is a birthplace of this thesis. But it's not one place. It's many.

It lies between Seoul and Sanaa. Between Budapest and Cairo. Philadelphia and DC. Edinburgh and Barcelona. Between Tehran and Seattle. Between Pardubice and Prague. Between Hadramawt and Isfahan. Between Vlčice and Chrudim.

It lies between here and all the countless places where I felt so happy within the past six years of my studies.

I would like to thank Barbora Vidová Hladká for encouraging me on my way. To Jan Hajič, Jarmila Panevová, Eva Hajičová, and Petr Sgall for their gentle personal guidance and the unlimited institutional support they provided me with.

I would like to thank Mark Liberman, Chris Cieri, Mohamed Maamouri, Hubert Jin, Kazuaki Maeda, David Graff, Wigdan Mekki, Dalila Tabessi, Farimah Partovi, and especially Ann Bies and Tim Buckwalter for welcoming me most warmly every time, and for having let me spend my Fulbright-Masaryk Fellowship at the Linguistic Data Consortium.

I would like to thank Petr Zemánek and my other friends and colleagues involved in the Prague Arabic Dependency Treebank project, as I am very grateful they share their interests with mine, and enjoy working with them.

Of the colleagues from the Institute of Formal and Applied Linguistics, I would like to thank Zdeněk Žabokrtský and Magda Ševčíková for letting me talk to them about anything anytime, and for their sense of humor. Let me thank also to Petr Pajas, Jan Štěpánek, Jiří Havelka, and Ondřej Bojar for many enjoyable and inspiring discussions, and to Libuše Brdičková and Marie Křížková for their kindness and care.

This thesis would not have developed into the present form were it not for the unique work of Tim Buckwalter, Markus Forsberg, Aarne Ranta, Peter Ljunglöf, and Ali El Dada, as well as Nizar Habash, Owen Rambow, and Khalil Sima'an. I would like to thank them very much for that, and for the interest and attention that they have paid to my work.

I am indebted to Benjamin Pierce for having taught me the elements of Haskell, and to Jiří Fleissig, Charif Bahbouh, Najet Boumellala, Cho Seung Kyu, Tomáš Horák, Kim Cheong Min, Zahwa Khira, Hamoud Hubaysh, and Dina Nourelahian for having amazed me with Oriental languages. Thank you very much.

I would like to thank my dear friend Hadoon Al-Attass and his family in Yemen for everything, and would like to greet You Hyun Jo and his family in Korea.

I would like to thank my parents for their endless love and support in all my life, and to all my family. And I would like to thank Iveta's parents for their kindest support and understanding, too.

I would like to thank my dear Iveta for creating the home for me, everywhere we went. As to this thesis, Iveta always listened carefully when I needed to discuss pieces of the work, shared my emotions and thoughts about it, and gave me her best response. I am happy she would just smile whenever I happened to be staring at Claude Monet's Water-Lily Pond and appeared being purely lazy, yet later blamed that on Haskell, calling it functional programming . . .

Otakar Smrž

Contents

List of Figures

I would not have made this so long
except that I do not have the leisure
to make it shorter.

Blaise Pascal, *Lettres Provinciales*

Chapter 1

Introduction

In this thesis, we are going to develop a computational model of the morphological processes in Arabic. With this system, we will become able to derive and inflect words, as well as analyze the structure of word forms and recognize their grammatical functions.

The approach to building our morphological model will strive to be comprehensive with respect to linguistic generalization, and high-level and modern with respect to the programming techniques that we will employ. We will describe the linguistic concept and will try to implement it in a very similar, yet abstract way, using the declarative functional programming language Haskell. We will promote the flexibility of our system, its reusability and extensibility.

1.1 Morphological Models

One can observe several different streams both in the computational and the purely linguistic modeling of morphology. Some are motivated by the need to analyze word forms as to their compositional structure, others consider word inflection as being driven by the underlying system of the language and the formal requirements of its grammar.

How do the current morphological analyzers of Arabic interpret, for instance, the number and gender of the masculine plurals *ǧudud* جُدُد 'new ones' or *quḍāh* قُضَاة 'judges', or the case of *mustawan* مُسْتَوًى 'a level'? Do they identify the values of these features that the syntax actually operates with, or is the resolution hindered by some too generic assumptions about the relation between meaning and form? What is the internal structure of the words? What lexemes or other word classes do they belong to?

There are substantial discrepancies between the grammatical descriptions of Arabic represented e.g. by (Fischer, 2001) or (Holes, 2004), and the information that the available morphological computational systems provide. One of the reasons is that there is never a complete consensus on what the grammatical description should be. The other source of the incompatibility lies in the observation that many implementations overlook the following general linguistic fact, restated in various contexts as the principal difference between the *function* and the *form* of a linguistic symbol:

> The morphosyntactic properties associated with an inflected word's individual inflectional markings may underdetermine the properties associated with the word as a whole. (Stump, 2001, p. 7)

According to Stump (2001), morphological theories can be classified along two scales. One of them deals with the question of inferability of meaning, and theories divide into:

incremental words *acquire* morphosyntactic properties only in connection with acquiring the inflectional exponents of those properties

realizational association of a set of properties with a word *licenses* the introduction of the exponents into the word's morphology

The other opposition concerns the core or the process of inflection:

lexical theories associate word's morphosyntactic properties with *affixes*

inferential theories consider inflection as a result of operations on *lexemes*; morphosyntactic properties are expressed by the *rules* that relate the form in a given paradigm to the lexeme

Evidence favoring inferential–realizational theories over the other three combinations is presented by Stump (2001) as well as Spencer (2004) or Baerman et al. (2006). We will discuss that in Chapter 3, presenting the risks that esp. the lexical–incremental approaches run into on concrete Arabic examples.

Many of the computational models of Arabic morphology, including in particular (Beesley, 2001), (Ramsay and Mansur, 2001) or (Buckwalter, 2002), are *lexical* in nature, i.e. they tend to treat inflectional affixes just like full-fledged lexical words. As they are not designed in connection with any syntax–morphology interface, their interpretations are destined to be *incremental*. That means that the only clue for discovering a word's morphosyntactic properties is through the explicit affixes and their prototypical functions.

Some signs of a *lexical–realizational* system can be found in (Habash, 2004). The author mentions and fixes the problem of underdetermination of inherent number with plurals, when developing a generative counterpart to (Buckwalter, 2002).

The computational models in (Cavalli-Sforza et al., 2000) and (Habash et al., 2005) attempt the *inferential–realizational* direction. Unfortunately, they implement only sections of the Arabic morphological system. The Arabic resource grammar in the Grammatical Framework (El Dada and Ranta, 2006) is perhaps the most complete inferential–realizational implementation to date. Its style is compatible with the linguistic description in e.g. (Fischer, 2001) or (Badawi et al., 2004), but the lexicon is now very limited and some other extensions for data-oriented computational applications are still needed.

ElixirFM, the implementation of the system developed in this thesis, is inspired by the methodology in (Forsberg and Ranta, 2004) and by functional programming, just like the

Arabic GF is (El Dada and Ranta, 2006). Nonetheless, ElixirFM reuses the Buckwalter lexicon (Buckwalter, 2002) and the annotations in the Prague Arabic Dependency Treebank (Hajič et al., 2004b), and implements a yet more refined linguistic model.

In our view, influenced by the Prague linguistic school and the theory of Functional Generative Description (Sgall et al., 1986, Sgall, 1967, Panevová, 1980, Hajičová and Sgall, 2003), the task of *morphology* should be to analyze word forms of a language not only by finding their internal structure, i.e. recognizing *morphs*, but even by *strictly* discriminating their functions, i.e. providing the true *morphemes*. This doing in such a way that it should be *completely* sufficient to generate the word form that represents a lexical unit and features all grammatical categories (and structural components) required by context, purely from the information comprised in the analyses.

It appears from the literature on most other implementations (many summarized in Al-Sughaiyer and Al-Kharashi, 2004) that the Arabic computational morphology has understood its role in the sense of operations with morphs rather than morphemes (cf. El-Sadany and Hashish, 1989), and has not concerned itself systematically and to the necessary extent with the role of morphology for syntax.[1] In other terms, the *syntax–morphology interface* has not been clearly established and respected.

The outline of formal grammar in (Ditters, 2001), for instance, works with grammatical categories like number, gender, humanness, definiteness, but one cannot see which of the existing systems could provide for this information correctly, as they misinterpret some morphs for bearing a category, and underdetermine lexical morphemes in general as to their intrinsic morphological functions. Nowadays, the only exception to this is the Arabic Grammatical Framework (El Dada and Ranta, 2006, Dada, 2007), which implements its own morphological and syntactic model.

Certain syntactic parsers, like (Othman et al., 2003), may resort to their own morphological analyzers, but still, they do not get rid of the form of an expression and only incidentally introduce truly functional categories. In syntactic considerations they often call for discriminative extra-linguistic features instead.[2] Commercial systems, e.g. (Chalabi, 2004), do not seem to overcome this interference either.

The missing common rationale as to what higher linguistic framework the computational morphology should serve for crystalizes in the number of individual, ad hoc feature sets and a very rare discussion of their motivation, completeness, relevance and actual expressive power. Even when addressing the tokenization problem of word forms tied together in the Arabic script, authors do not recognize the crucial mutual differences in their solutions, and do not define precisely the objectives of their models.

[1]Versteegh (1997b, chapter 6) describes the traditional Arabic understanding of *ṣarf* صَرف ‘morphology’ and *naḥw* نَحو ‘grammar, syntax’, where morphology studied the derivation of isolated words, while their inflection in the context of a sentence was part of syntax.

[2]Many people call those features *semantic* (cf. El-Shishiny, 1990), but we perceive them as *ontological*—our point is that those features are bound to some limited description of *reality*, not to the *linguistic* meaning itself.

1.2 Functional Approximation

Let us describe how the theoretical model of Functional Arabic Morphology, belonging to the *inferential–realizational* family, compares to the style of the Buckwalter Arabic Morphological Analyzer, classified as *lexical–incremental*. We will try to convert its information into the format of our model. The result will be called the *functional approximation*.

Buckwalter Arabic Morphological Analyzer (Buckwalter, 2002, 2004a) consists of a lexicon and a Perl program implementing an original algorithm for recognizing inflected Arabic words. It is the most widely used tool of its kind. The coverage of the lexicon is excellent (Buckwalter, 2004b, Maamouri and Bies, 2004) and the runtime performance of the program is very reasonable. Importantly enough, the first version of the Buckwalter analyzer was published as open-source software.

The analyzer consumes an ordinary Arabic text, resolves its contiguous orthographic *strings*, and produces morphological analyses characterizing each of them as a whole. The format of the output can be schematically depicted as follows:

```
(compositionofmorphs) [lemma_id]
  morph_1/tag_1 + morph_2/tag_2 + ... + morph_n/tag_n
```

The *morphs* group implicitly into the prefix, stem and suffix *segments*,[3] and the lemma identifies the semantically dominant morph, usually the stem, if there is one. Morphs are labeled with *tags* giving them the feel that they must be morphemes, which is the source of the disagreement between incremental and realizational interpretations, as noted earlier.

Let us illustrate these terms on a common example. Buckwalter's morphology on the string wbjAnbhA وبجانبها meaning 'and next to her' would yield

```
(wabijAnibihA) [jAnib_1]
  wa/CONJ + bi/PREP +                          prefix(es)
  jAnib/NOUN +                                 stem
  i/CASE_DEF_GEN + hA/POSS_PRON_3FS            suffix(es)
```

with the segments now indicated explicitly. The underlying *lexical words* or the syntactic *tokens*, as we denote them, are however still implicit. They read wa *wa-* وَ 'and', bi *bi-* بِ 'at', jAnib+i *ǧānib-i* جَانِبِ 'side-of' and hA *-hā* هَا 'her'. Note the morph i, which is a mere affix and not a run-on token, unlike the other three *clitics*.

There is *not enough functional information* provided in this kind of analyses, which we claim in Chapter 3. Yet, in the past experience with Buckwalter morphology (cf. Hajič et al., 2004b, 2005), we tried to *approximate* the functional views as closely as possible, and developed our tokenization and tag conversion algorithms (Smrž and Pajas, 2004).

[3] Some researchers use the term *segment* for what is called a *morph* here, and allow strings to decompose to multiple prefixes and suffixes, possibly including even cliticized lexical words.

When morphs are regrouped into tokens, their original tags form sequences (central column below) which map into a vector of values of grammatical categories. The tokens of our example will receive these converted, quasi-functional, positional[4] tags (left column):

C---------	wa	CONJ	وَ	*wa-*
P---------	bi	PREP	بِ	*bi-*
N-------2R	jAnib+i	NOUN+CASE_DEF_GEN	جَانِبِ	*ǧānib-i*
S----3FS2-	hA	POSS_PRON_3FS	هَا	*-hā*

The *positional notation* starts with the major and minor part-of-speech and proceeds through mood and voice up to person (position six), gender, number, case, and state. The values of the categories are unset, i.e. rendered with -, either if they are irrelevant for the particular part-of-speech and its refinement (positions one and two), or if there are no explicit data present in the original analysis, like no information on gender and number in jAnib+i. On the contrary, categories may be implied in parallel, cf. suffixed possessive pronouns being treated as regular pronouns, but in functional genitive (position nine). Some values can only be set based on other knowledge, which is the case of formal reduced definiteness, referred to also as state (position ten).

In Figure 3.1, this whole transformation is illustrated on a excerpt from a running text. Strings in the first column there break into tokens, as indicated by dotted lines, and for tokens, quasi-functional tags are derived. Glosses as well as token forms with hyphens for morph boundaries are shown in the two rightmost columns of the figure.

1.3 Reused Software

The ElixirFM implementation of Functional Arabic Morphology would not have come to being were it not for many open-source software projects that we could use during our work, or by which we got inspired.

ElixirFM and its lexicons are licensed under GNU General Public License and are available on http://sourceforge.net/projects/elixir-fm/, along with other accompanying software (MorphoTrees, Encode Arabic) and the source code of this thesis (ArabTEX extensions, TreeX).

1.3.1 Buckwalter Arabic Morphological Analyzer

The bulk of lexical entries in ElixirFM is extracted from the data in the Buckwalter lexicon (Buckwalter, 2002). Habash (2004) comments on the lexicon's internal format. We devised an algorithm in Perl using the morphophonemic patterns of ElixirFM that finds the roots

[4] Esp. in Arabic, position five is reserved. Similar notations have been used in various projects, notably the European Multext and Multext-East projects, for languages ranging from English to Czech to Hungarian.

and templates of the lexical items, as they are available only partially in the original, and produces the ElixirFM lexicon in customizable formats for Haskell and for Perl.

1.3.2 Functional Morphology Library

Functional Morphology (Forsberg and Ranta, 2004) is both a methodology for modeling morphology in a paradigmatic manner, and a library of purposely language-independent but customizable modules and functions for Haskell. It partly builds on the Zen computational toolkit for Sanskrit (Huet, 2002). Functional Morphology is also related to the Grammatical Framework, cf. (El Dada and Ranta, 2006) and `http://www.cs.chalmers.se/~markus/FM/`. Functional Morphology exists e.g. for Urdu (Humayoun, 2006).

1.3.3 TrEd Tree Editor

TrEd `http://ufal.mff.cuni.cz/~pajas/tred/` is a general-purpose graphical editor for trees and tree-like graphs written by Petr Pajas. It is implemented in Perl and is designed to enable powerful customization and macro programming. We have extended TrEd with the annotation mode for MorphoTrees, cf. Chapter 7.

1.4 Running ElixirFM

ElixirFM 1.0 is intended for use with the Hugs interactive interpreter of Haskell, available for a number of platforms via `http://haskell.org/hugs/`.

Download and install Hugs, or esp. the WinHugs graphical interface, on your system.

ElixirFM needs some language extensions that must be specified when Hugs is run. Remember to use the option `-98` when invoking Hugs, i.e. `hugs -98`. With WinHugs, note the menu File > Options.

Install Hugs and, to minimize any confusion, run it from the directory where your `ElixirFM.hs` is located. In the first Hugs session, the search path `-P .:../Encode:` should be defined, and overlapping instances allowed, as follows:

```
Hugs> :s -P .:../Encode:
Hugs> :s +o
Hugs> :l ElixirFM
```

In your future sessions, you only need to load the module `:l ElixirFM` directly. On success, the prompt will change to

```
ElixirFM>
```

upon which you are most welcome to read Chapters 4, 5, and 6.

It may be instructive to experiment with the settings of Hugs, to appreciate its interactive features. Please, consult the online documentation for both Hugs and ElixirFM.

1.5 Original Contributions

Let us list the most important and original contributions of this thesis:

A Recognition of functional versus illusory morphological categories, definition of a minimal but complete system of inflectional parameters in Arabic

B Morphophonemic patterns and their significance for the simplification of the model of morphological alternations

C Inflectional invariant and its consequence for the efficiency of morphological recognition in Arabic

D Intuitive notation for the structural components of words

E Conversion of the Buckwalter lexicon into a functional format resembling printed dictionaries

F ElixirFM as a general-purpose model of morphological inflection and derivation in Arabic, implemented with high-level declarative programming

G Abstraction from one particular orthography affecting the clarity of the model and extending its applicability to other written representations of the language

H MorphoTrees as a hierarchization of the process of morphological disambiguation

I Expandable morphological positional tags, restrictions on features, their inheritance

J Open-source implementations of ElixirFM, Encode Arabic, MorphoTrees, and extensions for ArabTeX

And now read out the crossword's
solution, I can't see it so well.
ŠHARDKP LITL NN OP 'N TRESBD
... is it in Arabic or what?

Hurvínek & Spejbl's *Crossword Puzzle*

Chapter 2

Writing & Reading Arabic

In the context of linguistics, morphology is the study of word forms. In formal language theory, the symbols for representing words are an inseparable part of the definition of the language. In human languages, the concept is a little different—an utterance can have multiple representations, depending on the means of communication and the conventions for recording it. An abstract computational morphological model should not be limited to texts written in one customary orthography.

Following (Beesley, 1997, 1998) on this issue, let us summarize our understanding of the present terms by emphasizing these characteristics:

orthography set of conventions for representing a language using an associated set of symbols

transcription alternative, phonetically or phonologically motivated representation of the language, possibly romanization

transliteration orthography with carefully substituted symbols, yet preserving the original orthographic conventions

encoding transliteration mapping the orthographic symbols into numbers implemented as characters or bytes

This chapter will explore the interplay between the genuine writing system and the transcriptions of Arabic. We will introduce in detail the ArabTEX notation (Lagally, 2004), a morphophonemic transliteration scheme adopted as the representation of choice for our general-purpose morphological model. We will then discuss the problem of recognizing the internal structure of words given the various possible types of their record.

2.1 Orthography and Buckwalter Transliteration

The standard Arabic orthography is based on the Arabic script. It has an alphabet of over 30 letters written from right to left in a cursive manner, and of about 10 other symbols written optionally as diacritics above or below the letters.

Phonologically, Arabic includes 28 consonants that are evenly divided into two groups according to their potential to assimilate with the definite article *al-*. They are called solar consonants *al-ḥurūf aš-šamsīya* and lunar consonants *al-ḥurūf al-qamarīya* in the linguistic tradition. Of the lunar consonants, *y*, *w*, and ʾ are weak in some contexts.

There are only six vocalic phonemes in Modern Standard Arabic, namely the short *a*, *i*, *u* and the long *ā*, *ī*, *ū*, the explicit spelling of which needs diacritics. In the dialects, the range of vowels is extended, but the script does not have any extra graphemes for encoding them. On the other hand, the diacritics also include symbols indicating an absence of a vowel after a consonant, or marking the indefinite article *-n* combined with the vocalic endings into distinct orthographic symbols.

Several letters of the script serve as allographs representing the single phoneme *hamza*, the glottal stop. Other allographs are motivated by morphophonemic changes or the historical development of the language and the script (Fischer, 2001, Holes, 2004, pages 3–34, resp. 89–95).

The set of letters is shown in Figure 2.1. The survey of conventions for using them and interleaving them with the diacritics will emerge shortly as a side-effect of our describing the ArabTEX notation.

Buckwalter transliteration (Buckwalter, 2002, 2004b) is a lossless romanization of the contemporary Arabic script, and is a one-to-one mapping between the relevant Unicode code points for Arabic and lower ASCII characters. It is part of Figure 2.1.[1]

2.2 ArabTEX Notation

The ArabTEX typesetting system (Lagally, 2004) defines its own Arabic script meta-encoding that covers both contemporary and historical orthography to an exceptional extent. The notation is human-readable and very natural to write with. Its design is inspired by the standard phonetic transcription of Arabic, which it mimics, yet some distinctions are introduced to make the conversion to the original script or the transcription unambiguous.

Unlike other transliteration concepts based on the strict one-to-one substitution of graphemes, ArabTEX interprets the input characters in context in order to get their proper meaning. Deciding the glyphs of letters (initial, medial, final, isolated) and their ligatures is not the issue of encoding, but of visualizing of the script. Nonetheless, definite article assimilation, inference of *hamza* carriers and silent *ʾalif*s, treatment of auxiliary vowels, optional quoting of diacritics or capitalization, resolution of notational variants, and mode-dependent processing remain the challenges for parsing the notation successfully.

ArabTEX's implementation is documented in (Lagally, 1992), but the parsing algorithm for the notation has not been published except in the form of the source code. The TEX code

[1]Buckwalter transliteration will be displayed in the `upright` typewriter font, whereas the ArabTEX notation will use the *`italic`* typewriter shape.

Lunar consonants

<table>
<tr><td>hamza</td><td>ʾ</td><td>'|</td><td>'</td><td>ء</td></tr>
<tr><td></td><td>b</td><td>b</td><td>b</td><td>ب</td></tr>
<tr><td></td><td>ǧ</td><td>^g</td><td>j</td><td>ج</td></tr>
<tr><td></td><td>ḥ</td><td>.h</td><td>H</td><td>ح</td></tr>
<tr><td></td><td>ḫ</td><td>_h</td><td>x</td><td>خ</td></tr>
<tr><td>ʿayn</td><td>ʿ</td><td>`</td><td>E</td><td>ع</td></tr>
<tr><td></td><td>ġ</td><td>.g</td><td>g</td><td>غ</td></tr>
<tr><td></td><td>f</td><td>f</td><td>f</td><td>ف</td></tr>
<tr><td></td><td>q</td><td>q</td><td>q</td><td>ق</td></tr>
<tr><td></td><td>k</td><td>k</td><td>k</td><td>ك</td></tr>
<tr><td></td><td>m</td><td>m</td><td>m</td><td>م</td></tr>
<tr><td></td><td>h</td><td>h</td><td>h</td><td>ه</td></tr>
<tr><td></td><td>w</td><td>w</td><td>w</td><td>و</td></tr>
<tr><td></td><td>y</td><td>y</td><td>y</td><td>ي</td></tr>
</table>

Solar consonants

t	t	t	ت
ṯ	_t	v	ث
d	d	d	د
ḏ	_d	*	ذ
r	r	r	ر
z	z	z	ز
s	s	s	س
š	^s	$	ش
ṣ	.s	S	ص
ḍ	.d	D	ض
ṭ	.t	T	ط
ẓ	.z	Z	ظ
l	l	l	ل
n	n	n	ن

Variants of ***ʾalif***

ʾalif	(*ā*)	A	A	ا
waṣla	'	"	{	ٱ

Suffix-only letters

ʾalif maqṣūra	(*ā*)	Y	Y	ى
tāʾ marbūṭa	(*t/h*)	T	p	ة

Variants of ***hamza***

<table>
<tr><td>madda</td><td>ʾā</td><td>'A</td><td>|</td><td>آ</td></tr>
<tr><td></td><td>ʾ</td><td>'a</td><td>O</td><td>أ</td></tr>
<tr><td></td><td>ʾ</td><td>'i</td><td>I</td><td>إ</td></tr>
<tr><td></td><td>ʾ</td><td>'w</td><td>W</td><td>ؤ</td></tr>
<tr><td></td><td>ʾ</td><td>'y</td><td>}</td><td>ئ</td></tr>
</table>

Non-Arabic consonants

p	p	P	پ
č	^c	J	چ
ž	^z	R	ژ
v	v	V	ڤ
g	g	G	گ

Figure 2.1 The letters *ḥurūf* حُرُوف of the Arabic orthography (extended with graphemes for some non-Arabic consonants) and their corresponding Buckwalter transliteration (the XML-friendly version), ArabTEX notation (in the mode with explicit *hamza* carriers), and phonetic transcription, listed in the right-to-left order.

is organized into deterministic-parsing macros, yet the complexity of the whole system makes consistent modifications or extensions by other users quite difficult.

We are going to describe our own implementations of the interpreter in Chapter 9, where we will show how to decode the notation and its proposed extensions. To encode the Arabic script or its phonetic transcription into the ArabTeX notation requires heuristic methods, if we want to achieve linguistically appropriate results.

2.2.1 Standard notation

Let us first deal with the notational conventions that the current version of ArabTeX supports (Lagally, 2004).

Our explanation will take the perspective of a child learning how the sounds are represented in the writing, rather than of a calligrapher trying to encrypt the individual graphemes into the notation. This may bring some difficulty to those who think of the language primarily through orthography and not phonology.

Phonemes The notation for consonants is listed in Figure 2.1. Short vowels are coded as `a`, `i` and `u`, the long ones `A`, `I` and `U`. We concatenate consonants and vowels in their natural order.

`darasa`	*darasa*	دَرَسَ	`darasa`	'he studied'
`sAfarat`	*sāfarat*	سَافَرَتْ	`saAfarato`	'she travelled'
`ya^glisUna`	*yağlisūna*	يَجْلِسُونَ	`yajolisuwna`	'they sit'
`kitAbuhA`	*kitābuhā*	كِتَابُهَا	`kitaAbuhaA`	'her book'
`.hAsUbI`	*ḥāsūbī*	حَاسُوبِي	`HaAsuwbiy`	'my computer'

Long vowels produce a combination of a diacritic and a letter in the script. Doubling of a consonant is indicated with the *šadda* ّ `~` diacritic, while no vowel after a consonant results in the *sukūn* ْ `o`. These rules interoperate, so `U` an `I` can often, even though not always, behave in the orthographic representation like `uw` and `iy` would.

`.sarra.ha`	*ṣarraḥa*	صَرَّحَ	`Sar~aHa`	'he explained'
`^gayyidUna`	*ğayyidūna*	جَيِّدُونَ	`jay~iduwna`	'good ones'
`qawIyayni`	*qawīyayni*	قَوِيَّيْنِ	`qawiy~ayoni`	'two strong ones'
`'adUwuhu`	*ʿadūwuhu*	عَدُوُّهُ	`Eaduw~uhu`	'his enemy'
`zuwwArunA`	*zuwwārunā*	زُوَّارُنَا	`zuw~aArunaA`	'our visitors'
`tuwAfiqu`	*tuwāfiqu*	تُوَافِقُ	`tuwaAfiqu`	'you agree'

The consonant `T` and the long vowel `Y` can only appear as final letters, otherwise the former changes to `t` and the latter to `A` or `ay`. Determination of the orthographic carrier for *hamza* is subject to complex rules, but phonologically, there is just one `'` consonant.

'as'ilaTu	*ʾasʾilatu*	أَسْئِلَةُ	Oaso}ilapu	'questions-of'
'as'ilatunA	*ʾasʾilatunā*	أَسْئِلَتُنَا	Oaso}ilatunaA	'our questions'
yarY yarAnI	*yarā yarānī*	يَرَى يَرَانِي	yaraY yaraAniy	'he sees (me)'
‘alY ‘alayhi	*ʿalā ʿalayhi*	عَلَى عَلَيْهِ	EalaY Ealayohi	'on (him)'

Articles The definite article al- is connected by a hyphen with the word it modifies. If assimilation is to take place, either the word's initial consonant is doubled, or the l of the article is replaced with that consonant directly.

al-qamaru	*al-qamaru*	اَلْقَمَرُ	Aaloqamaru	'the moon'
al-^s^samsu	*aš-šamsu*	اَلشَّمْسُ	Aal$~amosu	'the sun'
a^s-^samsu	*aš-šamsu*	اَلشَّمْسُ	Aal$~amosu	'the sun'
al-lawnu	*al-lawnu*	اَللَّوْنُ	Aall~awonu	'the color'
al-llawnu	*al-lawnu*	اَللَّوْنُ	Aall~awonu	'the color'
al-'alwAnu	*al-ʾalwānu*	اَلْأَلْوَانُ	AaloOalowaAnu	'the colors'

The indefinite article N must be distinguished by capitalization. Whether or not the orthography requires an additional silent *ʾalif*, need not be indicated explicitly.

baytuN	*baytun*	بَيْتٌ	bayotN	'a house *nom.*'
baytiN	*baytin*	بَيْتٍ	bayotK	'a house *gen.*'
baytaN	*baytan*	بَيْتًا	bayotFA	'a house *acc.*'
madInaTuN	*madīnatun*	مَدِينَةٌ	madiynapN	'a city *nom.*'
madInaTiN	*madīnatin*	مَدِينَةٍ	madiynapK	'a city *gen.*'
madInaTaN	*madīnatan*	مَدِينَةً	madiynapF	'a city *acc.*'

It is, however, possible to enforce a silent prolonging letter after an indefinite article (cf. Lagally, 2004). Most notably, it is used for the phonologically motivated ending aNY.

siwaNY	*siwan*	سِوًى	siwFY	'equality'
quwaNY	*quwan*	قُوًى	quwFY	'forces'
ma‘naNY	*maʿnan*	مَعْنًى	maEonFY	'meaning'

Extras The silent *ʾalif* also appears at the end of some verbal forms. It is coded UA if representing *ū*, and aWA or just aW if standing for *aw*.

katabUA	*katabū*	كَتَبُوا	katabuwA	'they wrote'
ya.sIrUA	*yaṣīrū*	يَصِيرُوا	yaSiyruwA	'that they become'
da‘aWA	*daʿaw*	دَعَوْا	daEawoA	'they called'

`tatamannaW`	*tatamannaw*	تَتَمَنَّوْا	`tataman~awoA`	'that you wish *pl.*'
`raW`	*raw*	رَوْا	`rawoA`	'do see *pl.*'
`insaW`	*insaw*	إِنْسَوْا	`AinosawoA`	'do forget *pl.*'

The phonological auxiliary vowels that are prefixed are preserved in the notation, yet, they can be elided in speech or turned into *waṣla* ٱ `{` in the script. The auxiliary vowels that are suffixed can be marked as such by a hyphen, if one prefers so.

`_dawU a^s-^sa'ni`	*dawū 'š-šaʾni*	ذَوُو ٱلشَّأْنِ	`*awuw {l$~aOoni`	ad a.
`qAla iyqa.z`	*qāla 'yqaẓ*	قَالَ ٱيْقَظْ	`qaAla {yoqaZo`	ad b.
`'an-i ismI`	*ʿan-i 'smī*	عَنِ ٱسْمِي	`Eani {somiy`	ad c.
`al-i-i^gtimA'u`	*al-i-'ǧtimāʿu*	ٱلِٱجْتِمَاعُ	`Aali{jotimaAEu`	ad d.

a. 'those concerned' b. 'he said wake up' c. 'about my name' d. 'the society'

The defective writing of the long *ā* is `_a`, of the short *u* it is `_U`. Other historical writings of long vowels can also be expressed in the standard notation. The description of `_i`, `_u`, `^A`, `^I`, `^U`, as well as `_aU`, `_aI`, `_aY`, is given in (Lagally, 2004, Fischer, 2001).

`l_akinna`	*lākinna*	لٰكِنَّ	`l`kin~a`	'however'
`h_a_dA`	*hāḏā*	هٰذَا	`h`*aA`	'this'
`_d_alika`	*ḏālika*	ذٰلِكَ	`*`lika`	'that'
`h_a'ulA'i`	*hāʾulāʾi*	هٰؤُلَاءِ	`h`WulaA'i`	'these'
`'_Ul_a'ika`	*ʾulāʾika*	أُولٰئِكَ	`Ouwl`}ika`	'those'

The dialectal pronunciation of vowels, namely *e*, *ē* and *o*, *ō*, can be reflected in the phonetic transcription only. In orthography, this makes no difference and the vowels are rendered as *i*, *ī* and *u*, *ū*, respectively. A real solution to the dialectal phonology might be more complex than this, but the approach of interpretable phonological notation is valid.

`sOha^g"`	*sōhağ*	سُوهَج	`suwhaj`	'Sohag'
`as-suwEs"`	*as-suwēs*	ٱلسُّوِيس	`Aals~uwiys`	'Suez'
`.hom.s"`	*ḥomṣ*	حُمْص	`HumoS`	'Homs'
`'omAn"`	*ʿomān*	عُمَان	`EumaAn`	'Oman'
`al-ma.greb"`	*al-maġreb*	ٱلْمَغْرِب	`Aalomagorib`	'Maghreb'

There are some other symbols in the notation that allow us to encode more information than is actually displayed in the script or its transcription. One of those is `"` suppressing the printing of a *sukūn*, like in the examples above, or of a vowel diacritic. Another is `|`, the invisible consonant useful in some tricky situations, and finally `B`, the *taṭwīl*, a filler for stretching the adjacent letters apart in the cursive script.

Words Due to the minute form of certain lexical words, the Arabic grammar has developed a convention to join them to the ones that follow or precede, thus making the whole concatenation a single orthographic word.

> Although by any criteria separate words, *wa* 'and', *fa* 'so', *bi* 'in, by, with' and *li* 'to, for' are written as if they were part of the word that follows them. Functionally similar words that are "heavier" monosyllables or bisyllabic, for example, *ʾaw* 'or', *fī* 'in', *ʿalā* 'on', are not so written. (Holes, 2004, p. 92)

This concatenation rule applies further to prefixed *ta* 'by *oath particle*', *sa* 'will *future marker*', *ka* 'like, as', *la* '*emph. part.*', as well as to suffixed personal pronouns in genitive and accusative, and variably to *mā*, *ma* 'what' (Fischer, 2001, p. 14).

The ArabTEX notation suggests that such words be hyphenated if prefixed and merely annexed if suffixed, to ensure proper inference of *hamza* carriers and *waṣla*s.

<table>
<tr><td>bi-bu.t'iN</td><td>bi-buṭʾin</td><td>بِبُطْءٍ</td><td>bibuTo'K</td><td>'slowly'</td></tr>
<tr><td>bi-intibAhiN</td><td>bi-'ntibāhin</td><td>بِٱنْتِبَاهٍ</td><td>bi{notibaAhK</td><td>'carefully'</td></tr>
<tr><td>bi-al-qalami</td><td>bi-'l-qalami</td><td>بِٱلْقَلَمِ</td><td>bi{loqalami</td><td>'with the pen'</td></tr>
<tr><td>fa-in.sarafa</td><td>fa-'nṣarafa</td><td>فَٱنْصَرَفَ</td><td>fa{noSarafa</td><td>'then he departed'</td></tr>
<tr><td>ka-'annanI</td><td>ka-ʾannanī</td><td>كَأَنَّنِي</td><td>kaOan~aniy</td><td>'as if I'</td></tr>
<tr><td>li-'ArA'ihi</td><td>li-ʾārāʾihi</td><td>لِآرَائِهِ</td><td>li|raA}ihi</td><td>'for his opinions'</td></tr>
<tr><td>sa-'u`.tIka</td><td>sa-ʾuʿṭīka</td><td>سَأُعْطِيكَ</td><td>saOuEoTiyka</td><td>'I will give you'</td></tr>
<tr><td>wa-lahu</td><td>wa-lahu</td><td>وَلَهُ</td><td>walahu</td><td>'and he has'</td></tr>
<tr><td>ka-_d_alika</td><td>ka-ḏālika</td><td>كَذٰلِكَ</td><td>ka*`lika</td><td>'as well, like that'</td></tr>
<tr><td>wa-fI-mA</td><td>wa-fī-mā</td><td>وَفِيمَا</td><td>wafiymaA</td><td>'and in what'</td></tr>
</table>

The cliticized *li* and *la* must be treated with exceptional care as they interact with the definite article and the letter after it. In the current version of the ArabTEX standard, this is not very intuitive, and we propose its improvement in the next subsection.

lil-'asafi	*lil-ʾasafi*	لِلْأَسَفِ	liloOasafi	'unfortunately'
lin-nawmi	*lin-nawmi*	لِلنَّوْمِ	liln~awomi	'for the sleep'
li-llaylaTi	*li-llaylati*	لِلَّيْلَةِ	lil~ayolapi	'for the night'
li-ll_ahi	*li-llāhi*	لِلّٰهِ	lil~`hi	'for God'

Modes One of the highlights of ArabTEX is its invention of an artful high-level notation for encoding the possibly multi-lingual text in a way that allows further *interpretation* by the computer program. In particular, the notation can be typeset in the original orthography of the language or in some kind of transcription, under one rendering convention or

another. These options are controlled by setting the interpretation environment, and no change to the data resources is required. Reseting of the options can be done even locally with individual words or their parts.

We leave the listing of all the options up to the ArabTeX manual (Lagally, 2004). Yet, let us illustrate the flexibility of the notation and, more importantly, of any system that shall reuse it. The vocalization degree tells what kinds of diacritics be overt in the script:

`bi-al-'arabIyaTi`	*bi-'l-ʿarabīyati*			'in Arabic'
`\fullvocalize`		بِٱلْعَرَبِيَّةِ	`bi{loEarabiy~api`	
`\vocalize`		بِالعَرَبِيَّةِ	`biAlEarabiy~api`	
`\novocalize`		بالعربيّة	`bAlErby~p`	

The interpretation of the notation is language-dependent. With `\setarab`, which is the default, the Arabic rules for determining the *hamza* carriers or silent *ʾalif*s are enforced. The `\setverb` option switches into the verbatim mode instead, c.f. Figure 2.1.

The style of the phonetic transcription is also customizable, and several schemes are predefined. While the Arabic script does not capitalize its letters, using the `\cap` sequence in the data would capitalize the very next letter in the transcription.

There are also options affecting some calligraphic nuances. With `\yahnodots`, for instance, every final glyph of *yāʾ* ي `y` appears as if *ʾalif maqṣūra* ى `Y` were in its place.

2.2.2 Extended notation

Distinguishing between specification and implementation, let us describe the extensions to the notation here, and design new or modified interpreters for it in Chapter 9.

Doubled vowels `ii` and `uu` are set equal to `I`, resp. `U`, in all contexts, just like `aa` turns into `A` in the standard. The distinction between `I` versus `iy` and `U` versus `uw` is retained.

`zuuwaarii`	*zūwārī*	زُوَّارِي	`zuw~aAriy`	'my visitors'
`zuwwaariy`	*zuwwāriy*	زُوَّارِي	`zuw~aAriy`	'my visitors'
`zUwArI`	*zūwārī*	زُوَّارِي	`zuw~aAriy`	'my visitors'

Suffix-only letters `T` and `Y` become treated as `t` and `A` automatically if occurring inside an orthographic word due to morphological operations, esp. suffixation of pronouns.

`laylaTAni`	*laylatāni*	لَيْلَتَانِ	`layolataAni`	'two nights'
`.hayATuN`	*ḥayātun*	حَيَاةٌ	`HayaApN`	'a life'
`.hayATI`	*ḥayātī*	حَيَاتِي	`HayaAtiy`	'my life'
`siwYhu`	*siwāhu*	سِوَاهُ	`siwaAhu`	'except for him'
`yarYnI`	*yarānī*	يَرَانِي	`yaraAniy`	'he sees me'

Underlying *ʾalif maqṣūra* can always be coded in the data with `Y`, yet would appear correctly as normal *ʾalif* if preceded by *yāʾ*, to avoid forms like *dunyā* دُنْيَى (Fischer, 2001, p. 8).

`dunyY`	*dunyā*	دُنْيَا	`dunoyaA`	'world'
`'a.hyY`	*ʾaḥyā*	أَحْيَا	`OaHoyaA`	'he revived'

Silent *ʾalif* spelled as `UW` or `uW` compares more easily with the other `aW` variant of the verbal ending, while supporting both the *ū* and *uw* style. Writing `uuW` is equivalent to `UW`.

`katabUW`	*katabū*	كَتَبُوا	`katabuwA`	'they wrote'
`katabuW`	*katabuw*	كَتَبُوا	`katabuwA`	'they wrote'
`katabuuW`	*katabū*	كَتَبُوا	`katabuwA`	'they wrote'

Defective writing of `_I` enables easy encoding of *miʾah* مِائَة 'hundred' and its derivatives in this archaic version, just like `_U` works for *ʾulāʾika* أُولٰئِكَ 'those'.

`m_I'aTuN`	*miʾatun*	مِائَةٌ	`miA}apN`	'a hundred'
`mi'aTuN`	*miʾatun*	مِئَةٌ	`mi}apN`	'a hundred'
`m_I'aTAni`	*miʾatāni*	مِائَتَانِ	`miA}ataAni`	'two hundred'
`tis`um_I'aTiN`	*tisʿumiʾatin*	تِسْعُمِائَةٍ	`tisoEumiA}apK`	'nine hundred'
`m_I'AtuN`	*miʾātun*	مِائَاتٌ	`miA}aAtN`	'hundreds'

Clitics interaction of the pattern `lV-al-`, `lV-al-CC`, `lV-aC-C`, and `lV-all`, where `V` stands for a short vowel (possibly quoted with `"` or simply omitted) and `C` represents a consonant, is resolved according to the rules of orthography. Joining *li* or *la* with *al-* is therefore not a burden on the notation, and all clitics can just be concatenated, cf. page 20.

`li-al-'asafi`	*li-'l-ʾasafi*	لِلْأَسَفِ	`liloOasafi`	'unfortunately'
`li-an-nawmi`	*li-'n-nawmi*	لِلنَّوْمِ	`liln~awomi`	'for the sleep'
`li-al-laylaTi`	*li-'l-laylati*	لِلَّيْلَةِ	`lil~ayolapi`	'for the night'
`li-al-l_ahi`	*li-'l-lāhi*	لِلّٰهِ	`lil~`hi`	'for God'
`al-la_dayni`	*al-laḏayni*	اَللَّذَيْنِ	`Aall~a*ayoni`	'those two who'
`li-al-la_dayni`	*li-'l-laḏayni*	لِلَّذَيْنِ	`lil~a*ayoni`	'for those two who'
`alla_dIna`	*allaḏīna*	اَلَّذِينَ	`Aal~a*iyna`	'the ones who'
`li-alla_dIna`	*li-'llaḏīna*	لِلَّذِينَ	`lil~a*iyna`	'for the ones who'

Extended options such as `\noshadda` for removing even ّ ٓ from the displayed script.

2.3 Recognition Issues

Arabic is a language of rich morphology, both derivational and inflectional. Due to the fact that the Arabic script does usually not encode short vowels and omits some other important phonological distinctions, the degree of morphological ambiguity is very high.

In addition to this complexity, Arabic orthography prescribes to concatenate certain word forms with the preceding or the following ones, possibly changing their spelling and not just leaving out the whitespace in between them. This convention makes the boundaries of lexical or syntactic units, which need to be retrieved as *tokens* for any deeper linguistic processing, obscure, for they may combine into one compact *string* of letters and be no more the distinct 'words'.

Thus, the problem of disambiguation of Arabic encompasses not only diacritization (discussed in Nelken and Shieber, 2005), but even tokenization, lemmatization, restoration of the structural components of words, and the discovery of their actual morphosyntactic properties, i.e. morphological tagging (cf. Hajič et al., 2005, plus references therein). These subproblems, of course, can come in many variants, and are partially coupled.

When inflected lexical words are combined, additional phonological and orthographic changes can take place. In Sanskrit, such euphony rules are known as external *sandhi*. Inverting *sandhi* is usually nondeterministic, but the method is clear-cut (Huet, 2003, 2005).

In Arabic, euphony must be considered at the junction of suffixed clitics, esp. pronouns in genitive or accusative. In the script, further complications can arise, some noted earlier in this chapter, some exemplified in the chart below:

`dirAsaTI`	*dirāsatī*	دِرَاسَتِي	→	`dirAsaTu |I`	*dirāsatu ī*	دِرَاسَةُ يِ
			→	`dirAsaTi |I`	*dirāsati ī*	دِرَاسَةِ يِ
			→	`dirAsaTa |I`	*dirāsata ī*	دِرَاسَةَ يِ
`mu'allimIya`	*muʿallimīya*	مُعَلِّمِيَّ	→	`mu'allimU |I`	*muʿallimū ī*	مُعَلِّمُو يِ
			→	`mu'allimI |I`	*muʿallimī ī*	مُعَلِّمِي يِ
`katabtumUhA`	*katabtumūhā*	كَتَبْتُمُوهَا	→	`katabtum hA`	*katabtum hā*	كَتَبْتُمْ هَا
`'i^grA'uhu`	*ʾiǧrāʾuhu*	إِجْرَاؤُهُ	→	`'i^grA'u hu`	*ʾiǧrāʾu hu*	إِجْرَاءُ هُ
`'i^grA'ihi`	*ʾiǧrāʾihi*	إِجْرَائِهِ	→	`'i^grA'i hu`	*ʾiǧrāʾi hu*	إِجْرَاءِ هُ
`'i^grA'ahu`	*ʾiǧrāʾahu*	إِجْرَاءَهُ	→	`'i^grA'a hu`	*ʾiǧrāʾa hu*	إِجْرَاءَ هُ
`li-al-'asafi`	*li-'l-ʾasafi*	لِلْأَسَفِ	→	`li al-'asafi`	*li al-ʾasafi*	لِ اَلْأَسَفِ

Modeling morphosyntactic constraints among the reconstructed tokens, such as that verbs be followed by accusatives and nominals by genitives, can be viewed as a special case of restrictions on the tokens' features, which we deal with in Chapter 7.[2]

[2]ElixirFM 1.0 can recognize only well-tokenized words. Implementing tokenization and the euphony rules is due in its very next version. Please, check `http://sf.net/projects/elixir-fm/` for the updates.

'All human beings are born free and equal in dignity and rights. They are endowed with reason and conscience and should act towards one another in a spirit of brotherhood.'

يُولَدُ جَمِيعُ آلنَّاسِ أَحْرَارًا مُتَسَاوِينَ فِي آلْكَرَامَةِ وَآلْحُقُوقِ.
وَقَدْ وُهِبُوا عَقْلًا وَضَمِيرًا وَعَلَيْهِمْ أَنْ يُعَامِلَ بَعْضُهُمْ بَعْضًا بِرُوحِ آلْإِخَاءِ.

yuwladu jamiyEu {ln~aAsi OaHoraArFA mutasaAwiyna fiy {lokaraAmapi wa{loHuquwqi.
waqado wuhibuwA EaqolAF waDamiyrFA waEalayohimo Oano yuEaAmila baEoDuhumo
baEoDFA biruwHi {loIixaA'i.

يولد جميع الناس أحرارا متساوين في الكرامة والحقوق.
وقد وهبوا عقلا وضميرا وعليهم أن يعامل بعضهم بعضا بروح الإخاء.

ywld jmyE AlnAs OHrArA mtsAwyn fy AlkrAmp wAlHqwq.
wqd whbwA EqlA wDmyrA wElyhm On yEAml bEDhm bEDA brwH AlIxA'.

Yūladu ğamīʿu 'n-nāsi ʾaḥrāran mutasāwīna fī 'l-karāmati wa-'l-ḥuqūqi. Wa-qad wuhibū ʿaqlan wa-ḍamīran wa-ʿalayhim ʾan yuʿāmila baʿḍuhum baʿḍan bi-rūḥi 'l-ʾiḫāʾi.

Júladu džamíᶜu an-nási 'aḥráran mutasávína fí al-karámati va-al-ḥuḳúḳi. Va-ḳad vuhibú ᶜaḳlan va-ḍamíran va-ᶜalajhim 'an juᶜámila baᶜḍuhum baᶜḍan bi-rúḥi al-'ichá'i.

\cap yUladu ^gamI`u an-nAsi 'a.hrAraN mutasAwIna fI al-karAmaTi wa-al-.huqUqi.
\cap wa-qad wuhibUW `aqlaN wa-.damIraN wa-`alayhim 'an yu`Amila ba`.duhum
ba`.daN bi-rU.hi al-'i_hA'i.

Figure 2.2 Universal Declaration of Human Rights, Article 1, in the following versions: English, fully vocalized versus non-vocalized Arabic (including Buckwalter transliteration), the standard ZDMG phonetic transcription versus its experimental Czech modification, and the source text in the ArabTeX notation. Inspired by Simon Ager's http://www.omniglot.com/.

Figure 2.2 illustrates the different written representations of Arabic. In the script, the diacritics, and even some instances of *hamza* indicated in red, can often be omitted in the normal usage. Some other kinds of orthographic variation are also common (Buckwalter, 2004b). Note that our morphological model is designed to understand all these 'languages'. Tokenization is exemplified in Figures 3.1, 7.1, and 7.2.

اَلعَرَبِيَّةُ بَحرٌ لَا سَاحِلَ لَهُ.
Al-ʿarabīyah baḥr lā sāḥila lahu.
Arabic is an ocean without a shore.

popular saying

Chapter 3

Morphological Theory

This chapter will define lexical words as the tokens on which morphological inflection proper will operate. We will explore what morphosyntactic properties should be included in the functional model. We will discuss the linguistic and computational views on inflectional morphology.

Later in this chapter, we will be concerned with Arabic morphology from the structural perspective, designing original morphophonemic patterns and presenting roots as convenient inflectional invariants.

3.1 The Tokenization Problem

In the previous chapter, we observed that the lexical words are not always apparent, neither in the Arabic orthography, nor in the transcriptions that we use.

Tokenization is an issue in many languages. Unlike in Korean or German or Sanskrit (cf. Huet, 2003), in Arabic there are clear limits to the number and the kind of tokens that can collapse into one orthographic string.[1] This idiosyncrasy may have lead to the prevalent interpretation that the *clitics*, including affixed pronouns or single-letter 'particles', are of the same nature and status as the derivational or inflectional affixes. In effect, cliticized tokens are often considered inferior to some central lexical morpheme of the orthographic string, which yet need not exist if it is only clitics that constitutes the string . . .

We think about the structure of orthographic words differently. In treebanking, it is essential that morphology determine the tokens of the studied discourse in order to provide the units for the syntactic annotation. Thus, it is *nothing but these units* that must be promoted to tokens and considered equal in this respect, irrelevant of how the tokens are realized in writing.

To decide in general between pure morphological affixes and the critical run-on syntactic units, we use the criterion of substitutability of the latter by its synonym or analogy that can occur isolated. Thus, if *hiya* هِيَ '*nom.* she' is a syntactic unit, then the suffixed

[1]Even if such rules differ in the standard language and the various dialects.

String	Token Tag	Buckwalter Morph Tags	Full Token Form	Token Gloss
سيخبرهم	F---------	FUT	سَ *sa-*	will
	VIIA-3MS--	IV3MS+IV+IVSUFF_MOOD:I	يُخبِرُ *yu-ḫbir-u*	he-notify
	S----3MP4-	IVSUFF_DO:3MP	هُم *-hum*	them
بذلك	P---------	PREP	بِ *bi-*	about/by
	SD----MS--	DEM_PRON_MS	ذٰلِكَ *ḏālika*	that
عن	P---------	PREP	عَن *ʿan*	by/about
طريق	N-------2R	NOUN+CASE_DEF_GEN	طَرِيقِ *ṭarīq-i*	way-of
الرسائل	N-------2D	DET+NOUN+CASE_DEF_GEN	اَلرَّسَائِلِ *ar-rasāʾil-i*	the-messages
القصيرة	A-----FS2D	DET+ADJ+NSUFF_FEM_SG+ +CASE_DEF_GEN	اَلقَصِيرَةِ *al-qaṣīr-at-i*	the-short
والإنترنت	C---------	CONJ	وَ *wa-*	and
	Z-------2D	DET+NOUN_PROP+ +CASE_DEF_GEN	اَلإِنتَرنِتِ *al-ʾinternet-i*	the-internet
وغيرها	C---------	CONJ	وَ *wa-*	and
	FN------2R	NEG_PART+CASE_DEF_GEN	غَيرِ *ġayr-i*	other/not-of
	S----3FS2-	POSS_PRON_3FS	هَا *-hā*	them

Figure 3.1 Tokenization of orthographic strings into tokens in *he will notify them about that through SMS messages, the Internet, and other means*, and the disambiguated morphological analyses providing each token with its tag, form and gloss (lemmas are omitted). Here, token tags are obtained from Buckwalter tags.

-hā هَا '*gen.* hers/*acc.* her' is tokenized as a single unit, too. If *sawfa* سَوفَ '*future marker*' is a token, then the prefixed *sa-* سَ, its synonym, will be a token. Definite articles or plural suffixes do not qualify as complete syntactic units, on the other hand.

The leftmost columns in Figure 3.1 illustrate how orthographic strings are tokenized in the Prague Arabic Dependency Treebank (Hajič et al., 2004b), which may in detail contrast to the style of the Penn Arabic Treebank (examples in Maamouri and Bies, 2004).

Discussions can be raised about the subtle choices involved in tokenization proper, or about what orthographic transformations to apply when reconstructing the tokens. Habash and Rambow (2005, sec. 7) correctly point out the following:

> There is not a single possible or obvious tokenization scheme: a tokenization scheme is an analytical tool devised by the researcher.

Different tokenizations imply different amount of information, and further influence the options for linguistic generalization (cf. Bar-Haim et al., 2005, for the case of Hebrew). We will resume this topic in Chapter 7 on MorphoTrees.

3.2 Functional and Illusory Categories

Once tokens are recognized in the text, the next question comes to mind—while concerned with the token forms, what morphosyntactic properties do they express?

In Figure 3.1, the Buckwalter analysis of the word *ar-rasā'il-i* اَلرَّسَائِلِ 'the messages' says that this token is a noun, in genitive case, and with a definite article. It does not continue, however, that it is also the actual plural of *risāl-ah* رِسَالَة 'a message', and that this logical plural formally behaves as feminine singular, as is the grammatical rule for every noun not referring to a human. Its congruent attribute *al-qaṣīr-at-i* اَلقَصِيرَةِ 'the short' is marked as feminine singular due to the presence of the *-ah* ةَ morph. Yet, in general, the mere presence of a morph does not guarantee its function, and vice versa.

What are the genders of *ṭarīq* طَرِيق 'way' and *al-'internet* اَلإِنتِرنِت 'the Internet'? Their tags do not tell, and *ṭarīq* طَرِيق actually allows either of the genders in the lexicon.

This discrepancy between the implementations and the expected linguistic descriptions compatible with e.g. (Fischer, 2001, Badawi et al., 2004, Holes, 2004) can be seen as an instance of the general disparity between inferential–realizational morphological theories and the lexical or incremental ones. Stump (2001, chapter 1) presents evidence clearly supporting the former methodology, according to which morphology needs to be modeled in terms of lexemes, inflectional paradigms, and a well-defined syntax–morphology interface of the grammar. At least these three of Stump's points of departure deserve remembering in our situation (Stump, 2001, pages 7–11):

> The morphosyntactic properties associated with an inflected word's individual inflectional markings may underdetermine the properties associated with the word as a whole.
>
> There is no theoretically significant difference between concatenative and non-concatenative inflection.
>
> Exponence is the only association between inflectional markings and morphosyntactic properties.

Most of the computational models of Arabic morphology are *lexical* in nature, i.e. they associate morphosyntactic properties with individual affixes regardless of the context of other affixes (cf. a different assumption in Roark and Sproat, 2007). As these models are not designed in connection with any syntax–morphology interface, their interpretation is destined to be *incremental*, i.e. the morphosyntactic properties are acquired only as a composition of the explicit inflectional markings. This cannot be appropriate for such a language as Arabic,[2] and leads to the series of problems that we observed in Figure 3.1.

[2] Versteegh (1997b, chapter 6, page 83) offers a nice example of how the supposed principle of 'one morph one meaning', responsible for a kind of confusion similar to what we are dealing with, complicated some traditional morphological views.

Functional Arabic Morphology endorses the inferential–realizational principles. It re-establishes the system of inflectional and inherent *morphosyntactic properties* (or *grammatical categories* or *features*, in the alternative naming) and discriminates precisely the senses of their use in the grammar. It also deals with *syncretism* of forms (cf. Baerman et al., 2006) that seems to prevent the resolution of the underlying categories in some morphological analyzers.

The syntactic behavior of *ar-rasā'il-i* اَلرَّسَائِلِ 'the messages' disclosed that we cannot dispense with a single category for number or for gender, but rather, that we should always specify the sense in which we mean it: [3]

functional category is for us the morphosyntactic property that is involved in grammatical considerations; we further divide functional categories into

logical categories on which agreement with numerals and quantifiers is based

formal categories controlling other kinds of agreement or pronominal reference

illusory category denotes the value derived merely from the morphs of an expression

Does the classification of the senses of categories actually bring new quality to the linguistic description? Let us explore the extent of the differences in the values assigned. It may, of course, happen that the values for a given category coincide in all the senses. However, promoting the illusory values to the functional ones is in principle conflicting:

1. Illusory categories are set only by a presence of some 'characteristic' morph, irrespective of the functional categories of the whole expression. If lexical morphemes are not qualified in the lexicon as to the logical gender nor humanness, then the logical number can be guessed only if the morphological stem of the logical singular is given along with the stem of the word in question. Following this approach implies interpretations that declare illusory feminine singular for e.g. *sād-ah* سَادَة 'men', *qād-ah* قَادَة 'leaders', *quḍ-āh* قُضَاة 'judges', *dakātir-ah* دَكَاتِرَة 'doctors' (all functional masculine plural), illusory feminine plural for *bāṣ-āt* بَاصَات 'buses' (logical masculine plural, formal feminine singular), illusory masculine dual for *'ayn-āni* عَيْنَانِ 'two eyes', *bi'r-āni* بِئْرَانِ 'two wells' (both functional feminine dual), or even rarely illusory masculine plural for *sin-ūna* سِنُونَ 'years' (logical feminine plural, formal feminine singular), etc.

2. If no morph 'characteristic' of a value surrounds the word stem and the stem's morpheme does not have the right information in the lexicon, then the illusory category remains unset. It is not apparent that *ḥāmil* حَامِل 'pregnant' is formal feminine singular while *ḥāmil* حَامِل 'carrying' is formal masculine singular, or that *ǧudud* جُدُد 'new'

[3] One can recall here the terms *ma'nawīy* مَعْنَوِيّ 'by meaning' and *lafẓīy* لَفظِيّ 'by expression' distinguished in the Arabic grammar. The logical and formal agreement, or *ad sensum* resp. grammatical, are essential abstractions (Fischer, 2001), yet, to our knowledge, implemented only recently in (El Dada and Ranta, 2006).

is formal masculine plural while *kutub* كُتُب 'books' is formal feminine singular. The problem concerns every nominal expression individually and pertains to some verbal forms, too. It is the particular issue about the internal/broken plural in Arabic, for which the illusory analyses do not reveal any values of number nor gender. It would not work easily to set the desired functional values by some heuristic, as this operation could only be conditioned by the pattern of consonants and vowels in the word's stem, and that can easily mislead, as this relation is also arbitrary. Consider the pattern in *ʿarab* عَرَب 'Arabs' (functional masculine plural) vs. *ǧamal* جَمَل 'camel' (functional masculine singular) vs. *qaṭaʿ* قَطَع 'stumps' (logical feminine plural, formal feminine singular), or that in *ǧimāl* جِمَال 'camels' (logical masculine plural, formal feminine singular) vs. *kitāb* كِتَاب 'book' (functional masculine singular) vs. *ʾināṯ* إِنَاث 'females' (logical feminine plural, formal feminine singular or plural depending on the referent), etc.

Functional Arabic Morphology enables the functional gender and number information thanks to the lexicon that can stipulate some properties as inherent to some lexemes, and thanks to the paradigm-driven generation that associates the inflected forms with the desired functions directly.

Another inflectional category that we discern for nominals as well as pronouns is case. Its functional values are nominative, genitive, and accusative. Three options are just enough to model all the case distinctions that the syntax–morphology interface of the language requires. The so-called oblique case is not functional, as long as it is the mere denotation for the homonymous forms of genitive and accusative in dual, plural and diptotic singular (all meant in the illusory sense, cf. Fischer, 2001, pages 86–96).

Neither do other instances of reduction of forms due to case syncretism need special treatment in our generative model, cf. Chapter 5. In a nutshell—if the grammar asks for an accusative of *maʿn-an* مَعنًى 'meaning', it does not care that its genitive and nominative forms incidentally look identical. Also note that case is preserved when a noun is replaced by a pronoun in a syntactic structure. Therefore, when we abstract over the category of person, we can consider even *ʾanā* أَنَا '*nom.* I', *-ī* ـِي '*gen.* mine', and *-nī* ـنِي '*acc.* me' as members of the pronominal paradigm of inflection in case.

The final category to revise with respect to the functional and illusory interpretations is definiteness. One issue is the logical definiteness of an expression within a sentence, the other is the formal use of morphs within a word, and yet the third, the illusory presence or absence of the definite or the indefinite article.

Logical definiteness is binary, i.e. an expression is syntactically either definite, or indefinite. It figures in rules of agreement and rules of propagation of definiteness (cf. the comprehensive study by Kremers, 2003).

Formal definiteness is independent of logical definiteness. It introduces, in addition to indefinite and definite, the reduced and complex definiteness values describing word

formation of *nomen regens* in construct states and logically definite improper annexations, respectively. In Chapter 5, we further formalize this category and refine it with two more values, *absolute* and *lifted*, and adopt the denotation of it as state. Let us give examples:

indefinite *ḥulwatu-n* حُلوَةٌ '*nom.* a-sweet', *Ṣanʿāʾa* صَنعَاءَ '*gen./acc.* Sanaa', *ḥurray-ni* حُرَّينِ '*gen./acc.* two-free', *tisʿū-na* تِسعُونَ '*nom.* ninety', *sanawāti-n* سَنَوَاتٍ '*gen./acc.* years'

definite *al-ḥulwatu* اَلحُلوَةُ '*nom.* the-sweet', *al-ḥurray-ni* اَلحُرَّينِ '*gen./acc.* the-two-free', *at-tisʿū-na* اَلتِّسعُونَ '*nom.* the-ninety', *as-sanawāti* اَلسَّنَوَاتِ '*gen./acc.* the-years'

reduced *ḥulwatu* حُلوَةُ '*nom.* sweet-of', *wasāʾili* وَسَائِلِ '*gen.* means-of', *wasāʾila* وَسَائِلَ '*acc.* means-of', *ḥurray* حُرَّي '*gen./acc.* two-free-in', *muḥāmū* مُحَامُو '*nom.* attorneys-of', *maʿānī* مَعَانِي '*nom./gen.* meanings-of', *sanawāti* سَنَوَاتِ '*gen./acc.* years-of'

complex *al-mutaʿaddiday-i 'l-luġāti* اَلمُتَعَدِّدَيِ اللُّغَاتِ '*gen./acc.* the-two-multiple-of the-languages, the two multilingual', *al-ḥulwatu 'l-i-'btisāmi* اَلحُلوَةُ الاِبتِسَامِ '*nom.* the-sweet-of the-smile, the sweet-smiled'[4]

Proper names and abstract entities can be logically definite while formally and illusorily indefinite: *fī Kānūna 'ṯ-ṯānī* فِي كَانُونَ الثَّانِي 'in January, the second month of Kānūn'. *Kānūna* كَانُونَ 'Kānūn' follows the diptotic inflectional paradigm, which is indicative of formally indefinite words. Yet, this does not prevent its inherent logical definiteness to demand that the congruent attribute *aṯ-ṯānī* اَلثَّانِي 'the-second' be also logically definite. *Aṯ-ṯānī* اَلثَّانِي 'the-second' as an adjective achieves this by way of its formal definiteness.

From the other end, there are adjectival construct states that are logically indefinite, but formally not so: *rafīʿu 'l-mustawā* رَفِيعُ المُستَوَى 'a high-level, high-of the-level'. *Rafīʿu* رَفِيعُ 'high-of' has the form that we call reduced, for it is the head of an annexation. If, however, this construct is to modify a logically definite noun, the only way for it to mark its logical definiteness is to change its formal definiteness to complex, such as in *al-masʾūlu 'r-rafīʿu 'l-mustawā* اَلمَسؤُولُ الرَّفِيعُ المُستَوَى 'the-official the-high-of the-level'. We can now inflect the phrase in number. Definiteness will not be affected by the change, and will ensure that the plural definite and complex forms do get distinguished: *al-masʾūlū-na 'r-rafīʿū 'l-mustawā* اَلمَسؤُولُونَ الرَّفِيعُو المُستَوَى 'the-officials the-highs-of the-level'.

In our view, the task of *morphology* should be to analyze word forms of a language not only by finding their internal structure, i.e. recognizing morphs, but even by *strictly* discriminating their functions, i.e. providing the true morphemes. This doing in such a way that it should be *completely* sufficient to generate the word form that represents a lexical unit and features all grammatical categories (and structural components) required by context, purely from the information comprised in the analyses. Functional Arabic Morphology is a model that suits this purpose.

[4] The dropped-ن-plus-ال cases of *al-ʾiḍāfah ġayr al-ḥaqīqīyah* الإِضَافَة غَير الحَقِيقِيَّة 'the improper annexation' clearly belong here (cf. Smrž et al., 2007, for how to discover more examples of this phenomenon).

3.3 Inflectional Morphology

In the main matter of his book, Stump (2001) presents his theory of Paradigm Function Morphology. Although our implementation of inflection is independent of this theory (cf. Forsberg and Ranta, 2004, El Dada and Ranta, 2006, Smrž, 2007), the central notion he formulates is useful for the later exposure of ElixirFM paradigms, too:

> A *paradigm function* is a function which, when applied to the root of a lexeme L paired with a set of morphosyntactic properties appropriate to L, determines the word form occupying the corresponding cell in L's paradigm. (Stump, 2001, p. 32)

The 'root' should not be understood in the sense of Semitic linguistics. Rather, it is the core lexical information associated with the lexeme and available to the inflectional rules. Spencer (2004) argues for a Generalized PFM, thanks to which the paradigm function can express much more than concatenative inflection, i.e. some kind of internal inflection or derivational morphology can be modeled.

There is some controversy about (Stump, 2001). Stump's arguments for preferring inferential–realizational theories, his notation of Paradigm Function Morphology, and the implementations using the DATR formalism (Evans and Gazdar, 1996, plus references therein) have been critically evaluated by some of the versed computational morphologists. Karttunen (2003) makes the point that DATR/KATR (Finkel and Stump, 2002) is not efficient enough for morphological recognition, and questions the clarity of the notations. He presents his implementation of verbal morphology of Lingala written in the xfst language (Beesley and Karttunen, 2003) for defining and computing with finite-state transducers, and essentially refuses to accept Stump's method as well as his classification of morphologies.

Roark and Sproat (2007) take even more examples from Stump's book, and propose that the linguistic distinctions between inferential vs. lexical and realizational vs. incremental are less important from the computational point of view. They reimplement some phenomena from Sanskrit, Swahili, and Breton using the lextools notation, compiled to transducers again (Sproat, 2006). They credit inferential–realizational views as possibly more suitable for generalizations and linguistic insight, but otherwise equate them with the other approaches:

> But no matter: what is clear is that whatever the descriptive merits of Stump's approach, it is simply a mistake to assume that this forces one to view morphology as realizational–inferential rather than, say lexical–incremental, at a mechanistic level. The two are no more than refactorizations of each other. (Roark and Sproat, 2007, online draft, chapter 3)

Unfortunately, though, it may be the case that Roark and Sproat (2007) actually assume different definitions than Stump (2001), as to how computationally strong 'transducer composition' and 'affix composition' actually are. It seems that we share with (Stump, 2001) the objections against any 'illusory' instances of analysis, which are, however, a property of the linguistic model, and not of the components of its implementation.

Roark and Sproat (2007, online draft, chapter 3) describe the ongoing debate as follows:

> As much as anything else, what is at issue here is not whether inferential–realizational or lexical–incremental theories are sometimes motivated, but whether they are always motivated, and whether there is any justification in the all-or-nothing view that has prevailed in morphological thinking.

We take from Stump's presentation the warnings about facts that may be 'risky' to accommodate properly in lexical or incremental approaches, which we have, after all, exemplified in the previous section.

Nonetheless, in our implementation, we use the programming techniques and formal notations that we find best suited for expressing the properties of the morphology that we wish to enforce with our model.

At the same time, we agree that one of the results of our model, namely the pairing of the valid morphological surface forms with their corresponding abstract representations, can as well be encoded in different ways, possibly using finite-state technology.

The advantage of our generic approach is that the morphological model is available in the programming language of our choice, with full abstractions that we can and would like to achieve. The reimplementations of the model, in any other formalisms, can be viewed as applications or translations thereof, and can be generated algorithmically (Forsberg and Ranta, 2004). After all, both (Forsberg and Ranta, 2004) and (Huet, 2002) directly show the co-existence of high-level models and their applications compiled into finite-state machines for efficient recognition of the modeled languages.

3.4 Neutralization vs. Uninflectedness

Syncretism is the morphological phenomenon that some words or word classes show instances of systematic homonymy.

With the notion of *paradigms* and *syncretism* in mind, one should ask what is the minimal set of combinations of morphosyntactic inflectional parameters that cover the inflectional variability.

For example, it seems fine in Arabic to request the pronoun for first person feminine singular, despite it being homonymous with first person masculine. Other homonymy occurs in first person dual and plural, irrespective of gender. However, homonymy does not occur with second person pronouns, for instance, and neither is gender neutralized

with the verbal or nominal predicates. That is, in the sentences *'anā ta'bānu* أَنَا تَعْبَانُ 'I am tired *masc.*' and *'anā ta'bānatun* أَنَا تَعْبَانَةٌ 'I am tired *fem.*', the gender of *'anā* أَنَا 'I' is made explicit through subject–predicate agreement. It is not the case that *'anā* أَنَا would have no gender, neither that it would have both.

Baerman et al. (2006, pp. 27–35) characterize neutralization and uninflectedness as cases of morphological syncretism distinguished by the ability of the context to 'demand' the morphosyntactic properties in question:

> Whereas *neutralization* is about syntactic irrelevance as reflected in morphology, *uninflectedness* is about morphology being unresponsive to a feature that is syntactically relevant. (Baerman et al., 2006, p. 32)

So what we have observed with first person pronouns is not neutralization, but rather uninflectedness. It is therefore right to allow parameter combinations such as those mentioned above, i.e. SP---1[MF]S1- *'anā* أَنَا 'I' and SP---1[MF][DP]1- *naḥnu* نَحْنُ 'we'.

On the other hand, one might claim that in Arabic the mood feature is neutralized with perfective or imperative verb stems, while it is distinguished with imperfective stems. Also, imperative stems are *fixed* as inherently active and second person, presumably.

Thus, in our morphological model, the shorthand verbal tag V--------- is expanded heterogeneously: with imperatives it becomes VC----[MF][SDP]--, with perfectives it is VP-[AP]-[123][MF][SDP]--, with imperfectives VI[ISJE][AP]-[123][MF][SDP]--.

3.5 The Pattern Alchemy

In Functional Arabic Morphology, patterns constitute the inventory of phonological melodies of words, regardless of the words' other functions. Morphophonemic patterns abstract from the consonantal *root*, which is often recognized or postulated on etymological grounds. Other types of patterns, like the decomposition into separate *CV patterns* and *vocalisms*, can be derived from the *morphophonemic patterns*. In the inverse view, roots and patterns can interdigitate or interlock together to form the morphological word stem.

The traditional terms are *ǧaḏr* جَذر for *root* and *wazn* وَزن for *pattern*. There are, however, multiple notions associated with either of these terms, as we discuss further. The prototypical root in Arabic linguistics is *f ' l* فعل, delivering the meaning of 'doing, acting'.

Fischer (2001) uses patterns that abstract away from the root, but can include even inflectional affixes or occasionally restore weak root consonants. For instance, we can find references to patterns like *'af'ala* for *'aḥsana* أَحْسَنَ 'he did right' or *'ahdā* أَهْدَى 'he gave', but *'af'alu* for *'a'lā* أَعْلَى 'higher'. In our model, the *morphophonemic pattern* pertains to the morphological stem and reflects its phonological qualities. Thus, our patterns become HaFCaL for *'aḥsana* أَحْسَنَ, while HaFCY for both *'ahdā* أَهْدَى and *'a'lā* أَعْلَى.

Holes (2004) distinguishes between 'morphosemantic' and 'morphosyntactic' patterns, but this distinction does not seem at issue here. More importantly, he uses the term 'pattern' in connection with denoting specific derivational classes in morphology, which we denote by Roman numerals as well, but call them *derivational forms* in that context.[5]

For the discussion of phonological processes with *weak* verb or noun *patterns*, with various examples on consonantal assimilation and analysis of *vocalization patterns*, consult (Beesley, 1998a). For discussing the consonant *gemination patterns* and extended derivational forms, see (Beesley, 1998b). In both works, the author uses the term 'morphophonemic' as 'underlying', denoting the patterns like `CuCiC` or `staCCaC` or `maCCuuC`. Yet, he also uses the term for anything but the surface form, cf. "an interdigitated but still morphophonemic stem" or "there may be many phonological or orthographical variations between these morphophonemic strings and their ultimate surface pronunciation or spelling" (Beesley, 1998a).

Eleven years earlier, twenty from today, Kay (1987) gives an account on finite-state modeling of the nonconcatenative morphological operations. He calls CV patterns as 'prosodic templates', both terms following (McCarthy, 1981). For further terminological explanations, cf. (Kiraz, 2001, pages 27–46).

We choose to build on morphophonemic patterns rather than on CV patterns and vocalisms. Words like *istaǧāb* إِسْتَجَاب 'to respond' and *istaǧwab* إِسْتَجْوَب 'to interrogate' have the same underlying VstVCCVC pattern, so information on CV patterns alone would not be enough to reconstruct the differences in the surface forms. Morphophonemic patterns, in this case `IstaFAL` and `IstaFCaL`, can easily be mapped to the hypothetical CV patterns and vocalisms, or linked with each other according to their relationship. Morphophonemic patterns deliver more information in a more compact way.

For the design of our morphophonemic patterns, we have two extreme options: either enforce the patterns in the non-assimilated and prototypical versions only, or define the patterns so that they reflect the result of any plausible phonemic transformations that might apply. We decide for the latter option, except for the case with the derivational Form VIII and Form VII, where we compute the assimilations at the `Ft` and `nF` boundaries on the fly during the `interlock` method, cf. Chapters 5 and 6. Such morphophonemic patterns are also directly indicative of any weak phonological behavior when involved in inflection or even derivation. Still, all pattern operations are most efficient.

With this approach, we are also given more precise control on the actual word forms—we explicitly confirm that the 'word' the pattern should create does undergo the implied transformations. One can therefore speak of 'weak patterns' rather than of 'weak roots'.

Let us recycle the notable quotation of the grammarian Ibn Jinnī, appearing in (Kiraz, 2001, p. 47) and itself taken from (Versteegh, 1997a), which might remind us not to break

[5]These derivational classes are analogous to *binyanim* in Hebrew linguistics (cf. Finkel and Stump, 2002). The confusion of the terms is understandable, as typically a class is represented by or referred to via a prototypical pattern. However, patterns are by themselves independent of the classes.

the balance between the abstract and the concrete, nor interchange them inappropriately, at least when considering the issue of patterns and their linguistic relevance:

> The underlying form of *qāma* 'stood up' is *qawama*. ... This had led some people to believe that such forms, in which the underlying level is not identical with the surface form, at one time were current, in the sense that people once used to say instead of *qāma zaydun*: *qawama zaydun* 'Zaid stood up'. ... This is not the case, on the contrary: these words have always had the form that you can see and hear now. Ibn Jinnī (932–1002), *al-Ḫaṣāʾiṣ*

The idea of pre-computing the phonological constraints within CV patterns into the 'morphophonemic' ones is present in (Yaghi and Yagi, 2004), but is applied to verbs only and is perhaps not understood in the sense of a primary or full-fledged representation. However, parallels with our definition of the patterns exist:

> Data [for stem generation were] designed initially with no knowledge of the actual patterns and repetitions that occur with morphophonemic and affix transformation rules. ... To optimize further, the transformation may be made on the morphological pattern itself, thus producing a sound surface form template. This procedure would eliminate the need to perform morphophonemic transformations on stems. ... A coding scheme is adopted that continues to retain letter origins and radical positions in the template so that this [optimization] will not affect [the author's model of] affixation. ... The surface form template can be rewritten as إ$h_F 2$تّ$h_M 0$َ$h_L 2$ى `AiF2t~aM0aL2Y`. This can be used to form stems such as إتَّدَى `Ait~adaY` by slotting the root ودي `wdy`. (Yaghi and Yagi, 2004, sec. 5)

Yaghi's templates are not void of root-consonant 'placeholders' that actually change under inflection, cf. $h_F 2$ $h_L 2$ indexed by the auxiliary integers to denote their 'substitutability'. The template, on the other hand, reflects some of the orthographic details and includes Form VIII assimilations that can be abstracted from, cf. esp. the تَّ `t~a` group.

With Functional Arabic Morphology, the morphophonemic pattern of *ittadā* إتَّدَى is simply `IFtaCY`, the root being *wdy* ودي. One of its inflected forms is `IFtaCY |<< "tumA"` *ittadaytumā* إتَّدَيْتُمَا 'the two of you accepted compensation', to follow again the example in (Yaghi and Yagi, 2004).[6] We describe the essence of this notation in Chapter 5.

CV templates are viewed from the perspective of moraic templates in the Prosodic Morphology (McCarthy and Prince, 1990a,b), later discussed by Kiraz (2001) within his development of a multitier nonlinear morphological model. An implementation of Arabic morphology derived from this work is presented in (Habash et al., 2005, Habash and

[6]With the root *wqy* وقي attested in (Fischer, 2001, Buckwalter, 2002), we have `IFtaCY |<< "tumA"` *ittaqaytumā* إتَّقَيْتُمَا 'the two of you bewared', or `IFtaCY |<< "UW"` *ittaqaw* إتَّقَوْا 'they bewared'.

Rambow, 2006). The moraic approach can be summarized as taking into account the internal structure and weight of syllables, distinguishing open and closed syllables as one opposition, and light versus heavy ones as another (cf. Bird and Blackburn, 1991).

Given that we can define a mapping from morphophonemic templates into prosodic or moraic templates, which we easily can, we claim that the prosodic study of the templates is separable from the modeling of morphology. Nonetheless, connections between both can still be pursued, and morphology should keep in mind to alter the patterns if phonology requires so. The morphophonemic information in the lexicon is rich enough to allow that, and the availability of the data in our computational model makes linguistic analyses of the lexicon, like those of broken plurals mentioned in (McCarthy and Prince, 1990a, Kiraz, 2001, Soudi et al., 2001), rather straightforward and empirically verifiable.

3.6 The Inflectional Invariant

In our approach, we define roots as sequences of consonants. In most cases, roots are triliteral, such as *k t b* كتب, *q w m* قوم, *d s s* دسس, *r ʾ y* رأي, or quadriliteral, like *d ḥ r ǧ* دحرج, *ṭ m ʾ n* طمأن, *z l z l* زلزل.

Roots in Arabic are, somewhat by definition, inflectional invariants. Unless a root consonant is weak, i.e. one of *y*, *w* or *ʾ*, and unless it assimilates inside a Form VIII pattern, then this consonant will be part of the inflected word form. This becomes apparent when we consider the repertoire and the nature of morphophonemic patterns.

The corollary is that we can effectively exploit the invariant during recognition of word forms. We can check the derivations and inflections of the identified or hypothesized roots only, and need not inflect the whole lexicon before analyzing the given inflected forms in question! This is a very fortunate situation for the generative model in ElixirFM.

While this seems the obvious way in which learners of Arabic analyze unknown words to look them up in the dictionary, it contrasts strongly with the practice in the design of computational analyzers, where finite-state transducers (Beesley and Karttunen, 2003), or analogously tries (Forsberg and Ranta, 2004, Huet, 2002), are most often used. Of course, other languages than Arabic need not have such convenient invariants.

I love you when you dance,
when you freestyle in trance,
so pure, such an expression.

Alanis Morissette, *So Pure*

Chapter 4

Impressive Haskell

Haskell is a purely functional programming language based on typed λ-calculus, with lazy evaluation of expressions and many impressive higher-order features. For a remedy to this definition, please visit http://haskell.org/.

Haskell was given its name after the logician Haskell Brooks Curry (1900–1982).

It is beyond the scope of this work to give any general, yet accurate account of the language. We will only overview some of its characteristics. Please, visit Haskell's website for the most appropriate introduction and further references. Textbooks include e.g. (Hudak, 2000) or (Daumé III, 2002–2006).

In Chapter 5, we exemplify and illustrate the features of Haskell step by step while developing ElixirFM. In Chapter 9, we present the implementation of a grammar of rewrite rules for Encode Arabic.

Types are distinct sets of uniquely identified values. Data types describe data structures, the function type `->` can be viewed as an encapsulated operation that would map input values to output values. In a rather coarse nutshell, types describe and control the space and time of computation.

Values can be defined on the symbolic level, and can be atomic or structured. Numbers, characters, lists of values, sets, finite maps, trees, etc. are all different data types.

```
data Language = Arabic | Korean | Farsi | Czech | English
data Family = Semitic | IndoEuropean | Altaic
data Answer = Yes | No | Web

isFamily :: Language -> Family -> Answer

isFamily Arabic Semitic = Yes
isFamily Czech  Altaic  = No
isFamily _      _       = Web
```

The structure of a program must conform to the type system, and conversely, types of expressions can be inferred from the structure of the program. The verification of this

important formal property is referred to as type checking. Type correctness guarantees the syntactic validity of a program, and is a prerequisite, not a guarantee, to its semantic correctness.

Polymorphism means that types can be parametrized with other types. The following implementation of lists is an example thereof: [1]

```
data List a = Item a (List a) | End
```

In other words, lists of some type `a` consist of an `Item` joining the value of type `a` with the rest of `List a`, which repeats until the `End`. Lists like these are homogeneous—all elements of a given list must have the same type `a`.

Universal and existential quantification over types is explained in (Pierce, 2002, chapters 23–24). While the former scheme is much more common, both ElixirFM and Encode Arabic make a little use of existential types (Jones, 1997) to achieve elegant heterogeneous collections where appropriate, cf. the `Wrap` and `Mapper` type constructors, respectively.[2]

Type classes allow us to define functions that can be overloaded depending on the types the functions are instantiated for.

```
class Encoding e where

    encode :: e -> [UPoint] -> [Char]
    decode :: e -> [Char] -> [UPoint]

instance Encoding ArabTeX where

    encode = ...
    decode = ...

instance Encoding Buckwalter where

    encode = ...
    decode = ...
```

Here, we declare that the `ArabTeX` and `Buckwalter` types belong to the class of `Encoding`s. That means that they provide functions that take the particular encoding of type `e` and convert a string of characters into a list of Unicode code points, or vice versa.

Type classes with functional dependencies (Jones, 2000) are another useful extension of Haskell, exemplified in the `Morphing` and `Parser` classes, Chapters 5 and 9.

For more on types, type classes, and generic programming, cf. (Hinze and Jeuring, 2003b,a) and (Wadler and Blott, 1989).

[1] In Haskell, lists are pre-defined and recognize the `:` and `[]` values instead of `Item` and `End`.

[2] In interpreters like Hugs, you can explore the definitions and the source code via the `:info`, `:find`, or `:edit` commands.

Monads are famous also due to their very useful application in parsing, esp. recursive descent parsing (Ljunglöf, 2002, plus references). Monadic parsers strongly resemble the declarative definition of rewrite rules in the specification of the grammar they implement.

It is 'impressive' that lists, and many other polymorphic data types, belong to the class of monads. This class is used for sequencing of computations—a seemingly imperative issue, yet completely declarative (Wadler, 1997)! Therefore, the `do` notation, which is seen in Chapter 9, can be applied to lists, too, and the list comprehension syntax translates directly to the essential monadic operations (Wadler, 1985, Hudak, 2000).

Um, ah, you mean, um
I think that's a sort of surreal masterpiece
It has a semi-autobiographical feel about it
A bit of a mixture, you mean, *mish-mash*
That's the word I was looking for

Queen in *Queen Talks*

Chapter 5

ElixirFM Design

Functional Arabic Morphology is a formulation of the Arabic inflectional system seeking the working interface between morphology and syntax. ElixirFM is its high-level implementation that reuses and extends the Functional Morphology library for Haskell (Forsberg and Ranta, 2004), yet in the language-specific issues constitutes our original work.

Inflection and derivation are modeled in terms of paradigms, grammatical categories, lexemes and word classes. The functional and structural aspects of morphology are clearly separated. The computation of analysis or generation is conceptually distinguished from the general-purpose linguistic model.

The lexicon of ElixirFM is designed with respect to abstraction, yet is no more complicated than printed dictionaries. It is derived from the open-source Buckwalter lexicon (Buckwalter, 2002) and is enhanced with information sourcing from the syntactic annotations of the Prague Arabic Dependency Treebank (Hajič et al., 2004b).

In Section 5.1, we survey some of the categories of the syntax–morphology interface in Modern Written Arabic, as described by Functional Arabic Morphology. In passing, we will introduce the basic concepts of programming in Haskell, a modern purely functional language that is an excellent choice for declarative generative modeling of morphologies, as Forsberg and Ranta (2004) have shown.

Section 5.2 will be devoted to describing the lexicon of ElixirFM. We will develop a so-called domain-specific language embedded in Haskell with which we will achieve lexical definitions that are simultaneously a source code that can be checked for consistency, a data structure ready for rather independent processing, and still an easy-to-read-and-edit document resembling the printed dictionaries.

In Section 5.3, we will illustrate how rules of inflection and derivation interact with the parameters of the grammar and the lexical information. We will demonstrate, also with reference to the Functional Morphology library (Forsberg and Ranta, 2004), the reusability of the system in many applications, including computational analysis and generation in various modes, exploring and exporting of the lexicon, printing of the inflectional paradigms, etc. Further interesting examples will be provided in Chapter 6.

5.1 Morphosyntactic Categories

Functional Arabic Morphology and ElixirFM re-establish the system of *inflectional* and *inherent* morphosyntactic properties (alternatively named grammatical categories or features) and distinguish precisely the senses of their use in the grammar.

In Haskell, all these categories can be represented as distinct data types that consist of uniquely identified values. We can for instance declare that the category of case in Arabic discerns three values, that we also distinguish three values for number or person, or two values of the given names for verbal voice:

```
data Case = Nominative | Genitive | Accusative

data Number = Singular | Dual | Plural

data Person = First | Second | Third

data Voice = Active | Passive
```

All these declarations introduce new enumerated types, and we can use some easily-defined methods of Haskell to work with them. If we load this (slightly extended) program into the interpreter,[1] we can e.g. ask what category the value `Genitive` belongs to (seen as the `::` type signature), or have it evaluate the list of the values that `Person` allows:

```
...? :type Genitive      ⟶   Genitive :: Case
...? enum :: [Person]    ⟶   [First, Second, Third]
```

Lists in Haskell are data types that can be parametrized by the type that they contain. So, the value `[Active, Active, Passive]` is a list of three elements of type `Voice`, and we can write this if necessary as the signature `:: [Voice]`. Lists can also be empty or have just one single element. We denote lists containing some type `a` as being of type `[a]`.

Haskell provides a number of useful types already, such as the enumerated boolean type or the parametric type for working with optional values:

```
data Bool = True | False
data Maybe a = Just a | Nothing
```

Similarly, we can define a type that couples other values together. In the general form, we can write

```
data Couple a b = a :-: b
```

introducing the value `:-:` as a container for some value of type `a` and another of type `b`.[2]

---

[1]Details on Hugs `http://haskell.org/hugs/` are given in Chapter 1.

[2]Infix operators can also be written as prefix functions if enclosed in `()`. Functions can be written as operators if enclosed in ` `` `. We will exploit this when defining the lexicon's notation.

Let us return to the grammatical categories. Inflection of nominals is subject to several formal requirements, which different morphological models decompose differently into features and values that are not always complete with respect to the inflectional system, nor mutually orthogonal. We will explain what we mean by revisiting the notions of state and definiteness in contemporary written Arabic.

To minimize the confusion of terms, we will depart from the formulation presented in (El Dada and Ranta, 2006). In there, there is only one relevant category, which we can reimplement as `State'`:

```
data State' = Def | Indef | Const
```

Variation of the values of `State'` would enable generating the forms *al-kitābu* اَلكِتَابُ def., *kitābun* كِتَابٌ indef., and *kitābu* كِتَابُ const. for the nominative singular of 'book'. This seems fine until we explore more inflectional classes. The very variation for the nominative plural masculine of the adjective 'high' gets *ar-rafīʿūna* اَلرَّفِيعُونَ def., *rafīʿūna* رَفِيعُونَ indef., and *rafīʿū* رَفِيعُو const. But what value does the form *ar-rafīʿū* اَلرَّفِيعُو, found in improper annexations such as in *al-masʾūlūna 'r-rafīʿū 'l-mustawā* اَلمَسؤُولُونَ الرَّفِيعُو المُستَوَى 'the-officials the-highs-of the-level', receive?

It is interesting to consult for instance (Fischer, 2001), where state has exactly the values of `State'`, but where the definite state `Def` covers even forms without the prefixed *al-* اَل article, since also some separate words like *lā* لَا 'no' or *yā* يَا 'oh' can have the effects on inflection that the definite article has. To distinguish all the forms, we might think of keeping state in the sense of Fischer, and adding a boolean feature for the presence of the definite article... However, we would get one unacceptable combination of the values claiming the presence of the definite article and yet the indefinite state, i.e. possibly the indefinite article or the diptotic declension.

Functional Arabic Morphology refactors the six different kinds of forms (if we consider all inflectional situations) depending on two parameters. The first controls prefixation of the (virtual) definite article, the other reduces some suffixes if the word is a head of an annexation. In ElixirFM, we define these parameters as type synonyms to what we recall:

```
type Definite = Maybe Bool
type Annexing = Bool
```

The `Definite` values include `Just True` for forms with the definite article, `Just False` for forms in some compounds or after *lā* لَا or *yā* يَا (absolute negatives or vocatives), and `Nothing` for forms that reject the definite article for other reasons.

Functional Arabic Morphology considers state as a result of coupling the two independent parameters:

```
type State = Couple Definite Annexing
```

Thus, the indefinite state `Indef` describes a word void of the definite article(s) and not heading an annexation, i.e. `Nothing :-: False`. Conversely, *ar-rafīʿū* اَلرَّفِيعُو is in the state `Just True :-: True`. The classical construct state is `Nothing :-: True`. The definite state is `Just _ :-: False`, where _ is `True` for El Dada and Ranta and `False` for Fischer. We may discover that now all the values of `State` are meaningful.[3]

Type declarations are also useful for defining in what categories a given part of speech inflects. For verbs, this is a bit more involved, and we leave it for Figure 5.2. For nouns, we set this algebraic data type:

```
data ParaNoun = NounS Number Case State
```

In the interpreter, we can now generate all 54 combinations of inflectional parameters for nouns. Note that these can be shown in the positional notation as well, cf. Chapter 6:

```
...? [ NounS n c s | n <- enum, c <- enum, s <- values ]  ⟶

     [ NounS Singular Nominative (Nothing :-: False),
       NounS Singular Nominative (Nothing :-: True),
       NounS Singular Nominative (Just True :-: False),
       ...
       NounS Dual Genitive (Just True :-: False),
       NounS Dual Genitive (Just True :-: True),
       NounS Dual Genitive (Just False :-: False),
       ...
       NounS Plural Accusative (Just False :-: False),
       NounS Plural Accusative (Just False :-: True) ]
```

The function `values` is analogous to `enum`, and both need to know their type before they can evaluate. The 'magic' is that the bound variables `n`, `c`, and `s` have their type determined by the `NounS` constructor, so we need not type anything explicitly. We used the list comprehension syntax to cycle over the lists that `enum` and `values` produce, cf. (Hudak, 2000, Wadler, 1985, Daumé III, 2002–2006).

5.2 ElixirFM Lexicon

Unstructured text is just a list of characters, or string:

```
type String = [Char]
```

Yet words do have structure, particularly in Arabic. We will work with strings as the superficial word forms, but the internal representations will be more abstract (and computationally more efficient, too).

[3] With `Just False :-: True`, we can annotate e.g. the 'incorrectly' underdetermined *rafīʿū* رَفِيعُو in *hum-u 'l-masʾūlūna rafīʿū 'l-mustawā* هُمُ المَسؤُولُونَ رَفِيعُو المُستَوَى 'they-are the-officials highs-of the-level', i.e. 'they are the high-level officials'.

```
|> "k t b" <| [

    FaCaL         `verb`  [ "write", "be destined" ]       `imperf`  FCuL,

    FiCAL         `noun`  [ "book" ]                       `plural`  FuCuL,

    FiCAL |< aT   `noun`  [ "writing" ],

    FiCAL |< aT   `noun`  [ "essay", "piece of writing" ] `plural`  FiCAL |< At,

    FACiL         `noun`  [ "writer", "author", "clerk" ] `plural`  FaCaL |< aT
                                                           `plural`  FuCCAL,

    FuCCAL        `noun`  [ "kuttab", "Quran school" ]     `plural`  FaCACIL,

    MaFCaL        `noun`  [ "office", "department" ]       `plural`  MaFACiL,

    MaFCaL |< Iy  `adj`   [ "office" ],

    MaFCaL |< aT  `noun`  [ "library", "bookstore" ]       `plural`  MaFACiL     ]
```

Figure 5.1 Entries of the ElixirFM lexicon nested under the root *k t b* كتب using morphophonemic templates.

The definition of *lexemes* can include the derivational *root and pattern* information if appropriate, cf. (Habash et al., 2005), and our model will encourage this. The surface word *kitāb* كِتَاب 'book' can decompose to the triconsonantal root *k t b* كتب and the morphophonemic pattern `FiCAL` of type `PatternT`:

```
data PatternT = FaCaL                 | FAL | FaCY    | FaCL
              | HaFCAL     | HACAL        | HaFCA'              | HACA'
              | FiCAL                     | FiCA'
              | FuCCAL     | FUCAL
              | TaFACuL                   | TaFACI
              | MustaFCaL  | {- ... -}              | MustaFaCL

              {- ... -}

    deriving (Eq, Enum, Show)
```

The **deriving** clause associates `PatternT` with methods for testing equality, enumerating all the values, and turning the names of the values into strings:

```
...? show FiCAL   ⟶   "FiCAL"
```

We choose to build on morphophonemic patterns rather than CV patterns and vocalisms. Words like *istaǧāb* اِستَجَاب 'to respond' and *istaǧwab* اِستَجوَب 'to interrogate' have

the same underlying VstVCCVC pattern, so information on CV patterns alone would not be enough to reconstruct the surface forms. Morphophonemic patterns, in this case `IstaFAL` and `IstaFCaL`, can easily be mapped to the hypothetical CV patterns and vocalisms, or linked with each other according to their relationship. Morphophonemic patterns deliver more information in a more compact way.

For ease of editing of the above enumeration, we can for instance arrange it into six columns that include the same kind of phonologically transformed patterns. We can in fact introduce functions that would classify the patterns explicitly with their underlying phonological information, as well as verify the completeness of their definition. This kind of checking seems beneficial, and desirably 'algebraic'.[4]

Of course, ElixirFM provides functions for properly interlocking the patterns with the roots, cf. the `interlock` method for details:

```
...? merge "k t b"  FiCAL    ⟶  "kitAb"
...? merge "^g w b" IstaFAL  ⟶  "ista^gAb"
...? merge "^g w b" IstaFCaL ⟶  "ista^gwab"
...? merge "s ' l"  MaFCUL   ⟶  "mas'Ul"
...? merge "z h r"  IFtaCaL  ⟶  "izdahar"
```

The *izdahar* إِزْدَهَر 'to flourish' case exemplifies that exceptionless assimilations need not be encoded in the patterns, but can instead be hidden in rules.

The whole generative model adopts the multi-purpose notation of ArabTEX (Lagally, 2004) as a meta-encoding of both the orthography and phonology. Therefore, instantiation of the `"'"` *hamza* carriers or other merely orthographic conventions do not obscure the morphological model. With Encode Arabic interpreting the notation, ElixirFM can at the surface level process the original Arabic script (non-)vocalized to any degree or work with some kind of transliteration or even transcription thereof.

Morphophonemic patterns represent the stems of words. The various kinds of abstract prefixes and suffixes can be expressed either as atomic values, or as literal strings wrapped into extra constructors:

```
data Prefix = Al | LA | Prefix String

data Suffix = Iy | AT | At | An | Ayn | Un | In | Suffix String

al = Al; lA = LA    -- function synonyms

aT = AT; ayn = Ayn; aN = Suffix "aN"
```

Affixes and patterns are arranged together via the `Morphs a` data type, where `a` is a triliteral pattern `PatternT` or a quadriliteral `PatternQ` or a non-templatic word stem `Identity` of type `PatternL`:

[4]We cannot resist noting, after http://wikipedia.org/, that the discipline of *algebra* derives its name from the treatise "The Compendious Book on Calculation by Completion and Balancing" written in Arabic by the Persian mathematician and astronomer al-Khwārezmī (780–850), also the father of *algorithm*.

```
data PatternL = Identity
data PatternQ = KaRDaS | KaRADiS {- ... -}

data Morphs a = Morphs a [Prefix] [Suffix]
```

The word *lā-silkīy* لَاسِلكِيّ 'wireless' can thus be decomposed as the root *s l k* سلك and the value `Morphs FiCL [LA] [Iy]`. Shunning such concrete representations, we define new operators `>|` and `|<` that denote prefixes, resp. suffixes, inside `Morphs a`:

```
...? lA >| FiCL |< Iy    ⟶    Morphs FiCL [LA] [Iy]
```

Implementing `>|` and `|<` to be applicable in the intuitive way required Haskell's multi-parameter type classes with functional dependencies (Jones, 2000):

```
class Morphing a b | a -> b where

    morph :: a -> Morphs b

instance Morphing (Morphs a) a where

    morph = id

instance Morphing PatternT PatternT where

    morph x = Morphs x [] []
```

The **instance** declarations ensure how the `morph` method would turn values of type `a` into `Morphs b`. Supposing that `morph` is available for the two types, `|<` is a function on `y :: a` and `x :: Suffix` giving a value of type `Morphs b`. The intermediate result of `morph y` is decomposed, and `x` is prepended to the stack `s` of the already present suffixes.

```
(|<) :: Morphing a b => a -> Suffix -> Morphs b

y |< x = Morphs t p (x : s)

    where Morphs t p s = morph y

(>|) :: Morphing a b => Prefix -> a -> Morphs b

x >| y = Morphs t (x : p) s

    where Morphs t p s = morph y
```

If it is strings that we need to prefix or suffix into `Morphs`, then two more operators can come handy:

```
(>>|) :: Morphing a b => String -> a -> Morphs b

x >>| y = Prefix x >| y

(|<<) :: Morphing a b => a -> String -> Morphs b

y |<< x = y |< Suffix x
```

With the introduction of patterns, their synonymous functions and the `>|` and `|<` operators, we have started the development of what can be viewed as a domain-specific language embedded in the general-purpose programming language. Encouraged by the flexibility of many other domain-specific languages in Haskell, esp. those used in functional parsing (Ljunglöf, 2002) or pretty-printing (Wadler, 2003), we may design the lexicon to look like e.g.

```
module Elixir.Data.Lexicon
import Elixir.Lexicon

lexicon = listing  {- lexicon's header -}

 |> {- root one -} <| [ {- Entry a -} ]

 |> {- root two -} <| [ {- Entry b -} ]

 -- other roots or word stems and entries
```

and yet be a verifiable source code defining a data structure that is directly interpretable. The meaning of the combinators `|>` and `<|` could be supplied via an external module `Elixir.Lexicon`, so is very easy to customize. The effect of these combinators might be similar to the `:` and `:-:` constructors that we met previously, but perhaps other data structures might be built from the code instead of lists and pairs. These are some of our current definitions in `Elixir.Lexicon`:

```
type Lexicon = [Wrap Nest]

type Root = String

data Nest    a   =   Nest Root [Entry a]      deriving Show

data Lexeme a    =   RE   Root (Entry a)      deriving Show

data Wrap a = WrapS  (a String)
            | WrapT  (a PatternT)
            | WrapQ  (a PatternQ)
            | WrapL  (a PatternL)             deriving Show
```

```
instance Wrapping PatternT  where  wrap             = WrapT
                                   unwrap (WrapT x) = x

instance Wrapping PatternQ  where  wrap             = WrapQ
                                   unwrap (WrapQ x) = x

instance Wrapping PatternL  where  wrap             = WrapL
                                   unwrap (WrapL x) = x

instance Wrapping String    where  wrap             = WrapS
                                   unwrap (WrapS x) = x

(<|) :: Wrapping a => Root -> [Entry a] -> Wrap Nest

(<|) r l = wrap (Nest r l)

(|>) :: [a] -> a -> [a]

(|>) x y = (:) y x
```

Individual entries can be defined with functions in a convenient notational form using ``. Infix operators can have different precedence and associativity, which further increases the options for designing a lightweight, yet expressive, embedded language.

In Figure 5.1, each entry reduces to a record of type `Entry PatternT` reflecting internally the lexeme's inherent properties. Consider one such reduction below. Functions like `plural` or `gender` or `humanness` could further modify the `Noun`'s default information:

```
...? FiCAL |< aT `noun` [ "writing" ]   ⟶

     noun (FiCAL |< aT) [ "writing" ]   ⟶

     Entry (Noun [] Nothing Nothing)
           (morph (FiCAL |< aT))
           [ "writing" ]                ⟶

     Entry (Noun [] Nothing Nothing)
           (Morphs FiCAL [] [AT])
           [ "writing" ]
```

The lexicon of ElixirFM is derived from the open-source Buckwalter lexicon (Buckwalter, 2002).[5] We devised an algorithm in Perl using the morphophonemic patterns of ElixirFM that finds the roots and templates of the lexical items, as they are available only partially in the original, and produces the lexicon in formats for Perl and for Haskell.

Information in the ElixirFM lexicon can get even more refined, by lexicographers or by programmers. Verbs could be declared via indicating their derivational verbal form (that

[5] Habash (2004) comments on the lexicon's internal format.

```
data Mood = Indicative | Subjunctive | Jussive | Energetic  deriving (Eq, Enum)
data Gender = Masculine | Feminine                          deriving (Eq, Enum)

data ParaVerb = VerbP       Voice Person Gender Number
              | VerbI Mood Voice Person Gender Number
              | VerbC                    Gender Number      deriving Eq


paraVerbC :: Morphing a b => Gender -> Number -> [Char] -> a -> Morphs b
paraVerbC g n i = case n of

            Singular    -> case g of    Masculine -> prefix i . suffix ""
                                        Feminine  -> prefix i . suffix "I"

            Plural      -> case g of    Masculine -> prefix i . suffix "UW"
                                        Feminine  -> prefix i . suffix "na"

            _           ->                           prefix i . suffix "A"
```

Figure 5.2 Excerpt from the implementation of verbal inflectional features and paradigms in ElixirFM.

would, still, reduce to some `Morphs a` value), and deverbal nouns and participles could be defined generically for the extended forms. The identification of patterns as to their derivational form is implemented easily with the `isForm` method:

```
data Form = I | II | III | IV {- .. -} XV

...? isForm VIII IFtaCaL                 ⟶  True
...? isForm II   TaKaRDuS                ⟶  True
...? filter (`isForm` MuFCI) [I ..]      ⟶  [IV]
```

Nominal parts of speech need to be enhanced with information on the inherent number, gender and humanness, if proper modeling of linguistic agreement in Arabic is desired.[6] Experiments with the Prague Arabic Dependency Treebank (Hajič et al., 2004b) show that this information can be learned from annotations of syntactic relations.

5.3 Morphological Rules

Inferential–realizational morphology is modeled in terms of paradigms, grammatical categories, lexemes and word classes. ElixirFM implements the comprehensive rules that draw the information from the lexicon and generate the word forms given the appropriate morphosyntactic parameters. The whole is invoked through a convenient `inflect`

[6] Cf. e.g. (El Dada and Ranta, 2006, Kremers, 2003).

method, which is instantiated for various combinations of types in ElixirFM, and is further illustrated in Chapter 6:

```
class Inflect m p where

    inflect :: (Template a, Rules a, Forming a,
                Morphing a a, Morphing (Morphs a) a) =>

               m a -> p -> [(String, [(Root, Morphs a)])]
```

The lexicon and the parameters determine the choice of paradigms. The template selection mechanism differs for nominals (providing plurals) and for verbs (providing all needed stem alternations in the extent of the entry specifications of e.g. Hans Wehr's dictionary), yet it is quite clear-cut!

Alternations of verbal stems between perfective vs. imperfective and active vs. passive rely on the `verbStems` method, instantiated for all pattern types, and on a simple boolean function `isVariant` which decides, based merely on the inflectional parameters, whether phonological variants of *weak* or *gemination patterns* are needed or not.

Below is the implementation of verbal stem inflection for verbs of Form X. Note that it is only here and in Form I that the actual root consonants need to be taken into account. In all other Forms, the morphophonemic patterns associated normally, i.e. even implicitly, with the entries in the lexicon provide all the required phonological information.

```
verbStems X r

    | let x = words r in if null x || length x > 2 && x !! 1 == x !! 2
                                then False
                                else head x `elem` ["w", "y"] = [

      (   Nothing,   IstaFCaL,   UstUCiL,    StaFCiL,   StaFCaL   ),
      (   Nothing,   IstaFCY,    UstUCI,     StaFCI,    StaFCY    )

      ]

    | otherwise = [

      (   Nothing,   IstaFCaL,   UstuFCiL,   StaFCiL,   StaFCaL   ),
      (   Just    (  IstaFaL,    UstuFiL,    StaFiL,    StaFaL    ),
                     IstaFAL,    UstuFIL,    StaFIL,    StaFAL    ),
      (   Nothing,   IstaFCY,    UstuFCI,    StaFCI,    StaFCY    ),
      (   Nothing,   IstaFY,     UstuFI,     StaFI,     StaFY     ), -- ista.hY
      (   Just    (  IstaFCaL,   UstuFCiL,   StaFCiL,   StaFCaL   ),
                     IstaFaCL,   UstuFiCL,   StaFiCL,   StaFaCL   )

      ]
```

This seems as a novel abstraction, not present in the grammars (Fischer, 2001, Holes, 2004, Badawi et al., 2004). It simplifies greatly the morphological model. Again, we credit this to the particular design of the morphophonemic patterns, cf. Chapter 3.

In Figure 5.2, the algebraic data type `ParaVerb` restricts the space in which verbs are inflected by defining three Cartesian products of the elementary categories: a verb can have `VerbP` perfect forms inflected in voice, person, gender, number, `VerbI` imperfect forms inflected also in mood, and `VerbC` imperatives inflected in gender and number only.[7]

The paradigm for inflecting imperatives, the one and only such paradigm in ElixirFM, is implemented as `paraVerbC`. It is a function parametrized by some particular value of gender `g` and number `n`. It further takes the initial imperative prefix `i` and the verbal stem (both inferred from the morphophonemic patterns in the lexical entry) to yield the inflected imperative form. Note the polymorphic type of the function, which depends on the following:

```
prefix, suffix :: Morphing a b => [Char] -> a -> Morphs b

prefix x y = Prefix x >| y
suffix x y = y |< Suffix x
```

If one wished to reuse the paradigm and apply it on strings only, it would be sufficient to equate these functions with standard list operations, without any need to reimplement the paradigm itself.

The definition of `paraVerbC` is simple and concise due to the chance to compose with `.` the partially applied `prefix` and `suffix` functions and to virtually omit the next argument. This advanced formulation may seem not as minimal as when specifying the literal endings or prefixes, but we present it here to illustrate the options that there are. An abstract paradigm can be used on more abstract types than just strings.[8] Inflected forms need not be merged with roots yet, and can retain the internal structure:

```
...? paraVerbC Feminine Plural "u" FCuL   ⟶   "u" >>| FCuL |<< "na"

...? merge "k t b" (Prefix "u" >| FCuL |< Suffix "na")   ⟶
```

fem.: `"uktubna"` *uktubna* اُكْتُبْنَ pl. 'write!'

```
...? [ merge "q r '" (paraVerbC g n "i" FCaL) | g <- values,
                                                n <- values ]   ⟶
```

masc.: `"iqra'"` *iqra'* اِقْرَأْ sg. `"iqra'A"` *iqra'ā* اِقْرَآ du. `"iqra'UW"` *iqra'ū* اِقْرَؤُوا pl. 'read!'

fem.: `"iqra'I"` *iqra'ī* اِقْرَئِي sg. `"iqra'A"` *iqra'ā* اِقْرَآ du. `"iqra'na"` *iqra'na* اِقْرَأْنَ pl.

[7] Cf. (Forsberg and Ranta, 2004, El Dada and Ranta, 2006).

[8] Cf. some morphology-theoretic views in (Spencer, 2004).

The highlight of the Arabic morphology is that the 'irregular' inflection actually rests in strictly observing some additional rules, the nature of which is phonological. Therefore, surprisingly, ElixirFM does not even distinguish between verbal and nominal word formation when enforcing these rules.[9] This reduces the number of paradigms to the prototypical 3 verbal and 5 nominal! Yet, the model is efficient.

The nominal paradigms of inflection discern five kinds of structural endings. The `paraMasculine` and `paraFeminine` are just names for the *illusory* plural endings *-ūn* ُون and *-āt* َات, `paraDual` is for the *illusory* dual ending *-ān* َان. None of these paradigms implies anything about the *functional* morphosyntactic gender and number! Note the types of the paradigms.

```
paraTriptote, paraDiptote, paraDual, paraMasculine, paraFeminine ::
    Morphing a b => Case -> Definite -> Annexing -> a -> Morphs b

paraTriptote c d a = case (c, d, a) of

        (Nominative, Nothing, False)    -> suffix "uN"
        (Genitive,   Nothing, False)    -> suffix "iN"
        (Accusative, Nothing, False)    -> suffix "aN"

        (Nominative, _ , _ )            -> suffix "u"
        (Genitive,   _ , _ )            -> suffix "i"
        (Accusative, _ , _ )            -> suffix "a"

paraDiptote c d a = case (c, d, a) of

        (Nominative, Nothing, False)    -> suffix "u"
        ( _ ,        Nothing, False)    -> suffix "a"

        (Nominative, _ , _ )            -> suffix "u"
        (Genitive,   _ , _ )            -> suffix "i"
        (Accusative, _ , _ )            -> suffix "a"

paraDual c d a = case (c, d, a) of

        (Nominative, _ , False)         -> suffix "Ani"
        ( _ ,        _ , False)         -> suffix "ayni"

        (Nominative, _ , True)          -> suffix "A"
        ( _ ,        _ , True)          -> suffix "ay"

paraMasculine c d a = case (c, d, a) of

        (Nominative, _ , False)         -> suffix "Una"
        ( _ ,        _ , False)         -> suffix "Ina"
```

[9] Cf. (Fischer, 2001, pp. 21–23, 30, 94–95, 135–139) and (Holes, 2004, pp. 112–114, 172).

```
        (Nominative, _ , True)          -> suffix "U"
        ( _ ,        _ , True)          -> suffix "I"

paraFeminine c d a = case (c, d, a) of

        (Nominative, Nothing, False)    -> suffix "uN"
        ( _ ,        Nothing, False)    -> suffix "iN"

        (Nominative, _ , _ )            -> suffix "u"
        ( _ ,        _ , _ )            -> suffix "i"
```

Given that the morphophonemic patterns already do reflect the phonological restrictions, the only places of further phonological interaction are the prefix boundaries and the junction of the last letter of the pattern with the very adjoining suffix. The rules are implemented with `->-` and `-<-`, respectively, and are invoked from within the `merge` function:

```
merge :: (Morphing a b, Template b) => [Char] -> a -> [Char]

(->-) :: Prefix -> [Char] -> [Char]
(-<-) :: Char -> Suffix -> [Char]

'I' -<- x = case x of   AT      -> "iyaT"
                        Iy      -> "Iy"
                        Un      -> "Una"
                        In      -> "Ina"

                        Suffix ""       -> "i"

                        Suffix "Una"    -> "Una"
                        Suffix "U"      -> "U"
                        Suffix "UW"     -> "UW"

                        Suffix "Ina"    -> "Ina"
                        Suffix "I"      -> "I"

                        Suffix x | x `elem` ["i",  "u"]  -> "I"
                                 | x `elem` ["iN", "uN"] -> "iN"

                                 | "n" `isPrefixOf` x ||
                                   "t" `isPrefixOf` x    -> "I" ++ x

                        _       -> "iy" ++ show x

'Y' -<- x = case x of   AT      -> "AT"
                        Iy      -> "awIy"
                        Un      -> "awna"
                        In      -> "ayna"
```

```
Suffix ""       -> "a"

Suffix "Una"    -> "awna"
Suffix "U"      -> "aw"
Suffix "UW"     -> "aW"

Suffix "Ina"    -> "ayna"
Suffix "I"      -> "ay"

Suffix x | x `elem` ["a",  "i",  "u"]  -> "Y"
         | x `elem` ["aN", "iN", "uN"] -> "aNY"

         | "at" `isPrefixOf` x         -> x

Suffix "a^gIy"  -> "a^gIy"

_        -> "ay" ++ show x
```

`(-<-)` is likewise defined when matching on `'A'`, `'U'`, and when not matching. `(->-)` implements definite article assimilation and occasional prefix interaction with weak verbs.

Nominal inflection is also driven by the information from the lexicon and by phonology. The reader might be noticing that the morphophonemic patterns and the `Morphs a` templates are actually extremely informative. We can use them as determining the inflectional class and the paradigm function, and thus we can almost avoid other unintuitive or excessive indicators of the kind of weak morphology, diptotic inflection, and the like.

5.4 Applications

The ElixirFM linguistic model and the data of the lexicon can be integrated into larger applications or used as standalone libraries and resources.

The language-independent part of the system could rest in the Functional Morphology library (Forsberg and Ranta, 2004). Among other useful things, it implements the compilation of the inflected word forms and their associated morphosyntactic categories into morphological analyzers and generators. The method used for analysis is deterministic parsing with tries, cf. also (Huet, 2002, Ljunglöf, 2002).

Nonetheless, ElixirFM provides its original analysis method exploiting the inflectional invariant defined in Chapter 3. We can, at least in the present version of the implementation, dispense with the compilation into tries, and we use rather minimal computational resources.

We define a class of types that can be `Resolve`d, which introduces one rather general method `resolveBy` and one more specific method `resolve`, for which there is a default implementation. It says that the form in question should be resolved by equality `(==)`

with the inflected forms in the model. The generic `resolveBy` method can be esp. used for recognition of partially vocalized or completely non-vocalized representations of Arabic, or allow in fact arbitrary kinds of omissions. We illustrate this in Chapter 6.

```
class Eq a => Resolve a where

    resolveBy :: (String -> String -> Bool) -> a -> [[Wrap Token]]
    resolve   ::                               a -> [[Wrap Token]]

    resolve = resolveBy (==)

data Token a = Token { lexeme :: Lexeme a,
                       struct :: (Root, Morphs a),
                       tag    :: String }           deriving Show

data Entry a = Entry { entity :: Entity a,
                       morphs :: Morphs a,
                       reflex :: Reflex }           deriving Show

type Reflex = [String]
```

The `Token a` type refers to the complete information on the underlying lexeme, as well as describes the structure of the inflected form and its positional morphological tag, cf. Chapter 7. In contrast, `Entry a` types constitute the complete entries in the `Lexicon`, except that they inherit the information on `Root` which is shared in the whole `Nest a`, but which can be overridden. The `Entity a` record covers the inherent grammatical information. `Reflex` can be the lexical definition of the entry, i.e. in particular the English gloss extracted from the Buckwalter lexicon.

Reusing and extending the original Functional Morphology library, ElixirFM also provides functions for exporting and pretty-printing the linguistic model into XML, LaTeX, Perl, SQL, and other custom formats.

We have presented ElixirFM as a high-level functional implementation of Functional Arabic Morphology. Next to some theoretical points, we proposed a model that represents the linguistic data in an abstract and extensible notation that encodes both *orthography* and *phonology*, and whose interpretation is customizable. We developed a domain-specific language in which the lexicon is stored and which allows easy manual editing as well as automatic verification of consistency. We believe that the modeling of both the *written* language and the *spoken* dialects can share the presented methodology.

Only bad boys use such recursive calls,
but only good girls use this package.
Thus the problem is of minor interest.

Carsten Heinz, the listings package

Chapter 6

Other Listings

This chapter is a non-systematic overview of the features of ElixirFM. It can serve as a tutorial for the first sessions with ElixirFM in the environment of the Hugs interpreter.

For information on installation and the initial settings, cf. Chapter 1.

Note that the order of queries is not significant for the results. Unless, of course, different Hugs settings apply.

The repertoire of the user functions may be somewhat preliminary, and will be extended as practice requires. The function names that appear bold, thanks to the listings package of LaTeX, are implemented in standard Haskell, and can as well be explored using `:info` or **`:find`**.

```
Hugs> :l ElixirFM

ElixirFM> :f $
ElixirFM> :f .

ElixirFM> :i lexicon
lexicon :: Lexicon

ElixirFM> countNest lexicon
4290

ElixirFM> countEntry lexicon
13429
```

The lexicon that is normally loaded is defined in the `Elixir.Data.Lexicons` module. It is an abridged version of the ElixirFM lexicon. Working with it is faster, and it should cover most of the newswire text vocabulary. To access the complete lexicon, we need to `:load`, and **`import`** in the source code, the `Elixir.Data.Buckwalter` module instead.

```
ElixirFM> :e ElixirFM.hs
ElixirFM> :e FM/Arabic/Dictionary.hs

ElixirFM> :e Elixir/Data/Lexicons.hs
ElixirFM> :e Elixir/Data/Buckwalter.hs
```

```
ElixirFM> :l Elixir.Data.Buckwalter

Elixir.Data.Buckwalter> countNest lexicon
9916

Elixir.Data.Buckwalter> countEntry lexicon
40134

Elixir.Data.Buckwalter> :l ElixirFM
```

Let us find out what the **Show** class and the `:s -u` and `:s +u` options are responsible for:

```
ElixirFM> :s -u

ElixirFM> lA >| FiCL |< Iy
Morphs FiCL [LA] [Iy]

ElixirFM> IFtaCY |<< "UW"
Morphs IFtaCY [] [Suffix "UW"]

ElixirFM> :s +u

ElixirFM> lA >| FiCL |< Iy
lA >| FiCL |< Iy

ElixirFM> IFtaCY |<< "UW"
IFtaCY |<< "UW"

ElixirFM> IFtaCY |< Suffix "UW"
IFtaCY |<< "UW"
```

Some Arabic linguistic preliminaries:

```
ElixirFM> enum :: [PatternT]
[FaCaL,FAL,FaCY,FaCL,FaCA,FaCiL,FaCI,FaCuL,FaCU,FuCiL,...
...,IFCanLY,UFCunLY,FCanLI,FCanLY,IFCinLA',MuFCanLI,MuFCanLY]

ElixirFM> [ unwraps root x | x <- lexicon ]
...

ElixirFM> (take 10 . drop 50) [ unwraps root x | x <- lexicon ]
["' .t l","' _d _d","' _d '","' ^s r","' ^g w d",
"' ^g r","' _h '","' _d y","' _d r","' _d n"]
```

Simple lookup on the nests and entries of the `[Wrap Nest]` that you explicitly provided it with:

```
ElixirFM> :i lookupRoot
lookupRoot :: Root -> Lexicon -> [Wrap Nest]

ElixirFM> :i lookupEntry
```

```
lookupEntry :: String -> Lexicon -> [Wrap Lexeme]

ElixirFM> :i lookupLemma
lookupLemma :: String -> Lexicon -> String

ElixirFM> lookupRoot "^g _d r" lexicon
[WrapT (Nest "^g _d r" [Entry {entity = Noun [Right FuCUL,Right FaCL]
Nothing Nothing, morphs = FiCL, reflex = ["root","stub"]},Entry
{entity = Adj [] Nothing, morphs = FiCL |< Iy, reflex = ["radical","root"]}])]

ElixirFM> lookupEntry "kitAb" lexicon
[WrapT (RE "k t b" Entry {entity = Noun [Right FuCuL] Nothing Nothing,
morphs = FiCAL, reflex = ["book"]})]

ElixirFM> lookupLemma "kitAb" lexicon
"kitAb (k t b) FiCAL\n\tNoun [Right FuCuL] Nothing Nothing\nbook\n\n"

ElixirFM> putStr $ lookupLemma "kitAb" lexicon
kitAb (k t b) FiCAL
        Noun [Right FuCuL] Nothing Nothing
book
```

Exploration of the `inflect` method. Note that it works on the information that you give it. To ensure that the lexical information is correct, provide it with the results of `lookupEntry`.

```
ElixirFM> inflect (FiCAL `noun` []) "--------2-"
[("N------S2I",[("f ' l",FiCAL |<< "iN")]),("N------S2R",[("f ' l",...

ElixirFM> pretty $ inflect (FiCAL `noun` []) "--------2-"
("N------S2I",[("f ' l",FiCAL |<< "iN")])
("N------S2R",[("f ' l",FiCAL |<< "i")])
("N------S2D",[("f ' l",al >| FiCAL |<< "i")])
("N------S2C",[("f ' l",al >| FiCAL |<< "i")])
("N------S2A",[("f ' l",FiCAL |<< "i")])
("N------S2L",[("f ' l",FiCAL |<< "i")])
("N------D2I",[("f ' l",FiCAL |<< "ayni")])
("N------D2R",[("f ' l",FiCAL |<< "ay")])
("N------D2D",[("f ' l",al >| FiCAL |<< "ayni")])
("N------D2C",[("f ' l",al >| FiCAL |<< "ay")])
("N------D2A",[("f ' l",FiCAL |<< "ayni")])
("N------D2L",[("f ' l",FiCAL |<< "ay")])
("N------P2I",[])
("N------P2R",[])
("N------P2D",[])
("N------P2C",[])
("N------P2A",[])
("N------P2L",[])

ElixirFM> pretty $ inflect (RE "k t b" $ FiCAL `noun` []) "-------S2[IDR]"
```

```
("N------S2I",[("k t b",FiCAL |<< "iN")])
("N------S2R",[("k t b",FiCAL |<< "i")])
("N------S2D",[("k t b",al >| FiCAL |<< "i")])

ElixirFM> uncurry merge ("k t b", FiCAL |<< "iN")
"kitAbiN"

ElixirFM> pretty $ inflect (RE "k t b" $ FiCAL `noun` [] `plural` FuCuL)
                           "-------P2[IDR]"
("N------P2I",[("k t b",FuCuL |<< "iN")])
("N------P2R",[("k t b",FuCuL |<< "i")])
("N------P2D",[("k t b",al >| FuCuL |<< "i")])
```

Resolving well-tokenized inflected forms:

```
ElixirFM> pretty $ resolveBy (omitting "aiuAUI") "ktbuN"
N------S1I kitAbuN "k t b" FiCAL ["book"]
N------P1I kutubuN "k t b" FiCAL ["book"]
N------S1I kAtibuN "k t b" FACiL ["writer","author","clerk"]
A-----MS1I kAtibuN "k t b" FACiL ["writing"]

ElixirFM> pretty $ resolveBy (omitting "aiuAUI") "’ArA’"
N------P1R ’ArA’u "r ’ y" FaCL ["opinion","view","idea"]
N------P1A ’ArA’u "r ’ y" FaCL ["opinion","view","idea"]
N------P1L ’ArA’u "r ’ y" FaCL ["opinion","view","idea"]
N------P2R ’ArA’i "r ’ y" FaCL ["opinion","view","idea"]
N------P2A ’ArA’i "r ’ y" FaCL ["opinion","view","idea"]
N------P2L ’ArA’i "r ’ y" FaCL ["opinion","view","idea"]
N------P4R ’ArA’a "r ’ y" FaCL ["opinion","view","idea"]
N------P4A ’ArA’a "r ’ y" FaCL ["opinion","view","idea"]
N------P4L ’ArA’a "r ’ y" FaCL ["opinion","view","idea"]

ElixirFM> pretty $ resolveBy (omitting $ (encode UCS . decode Tim) "~aiuKNF")
                             (decode Tim "lAslky")
A-----MS1I lA-silkIyuN "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS1R lA-silkIyu "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS1A lA-silkIyu "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS1L lA-silkIyu "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS2I lA-silkIyiN "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS2R lA-silkIyi "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS2A lA-silkIyi "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS2L lA-silkIyi "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS4R lA-silkIya "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS4A lA-silkIya "s l k" lA >| FiCL |< Iy ["wireless","radio"]
A-----MS4L lA-silkIya "s l k" lA >| FiCL |< Iy ["wireless","radio"]

ElixirFM> pretty $ resolveBy (omitting $ (encode UCS . decode Tim) "~aiuKNF")
                             (decode Tim "ktAb")
N------S1I kitAbuN "k t b" FiCAL ["book"]
N------S1R kitAbu "k t b" FiCAL ["book"]
```

```
N------S1A kitAbu "k t b" FiCAL ["book"]
N------S1L kitAbu "k t b" FiCAL ["book"]
N------S2I kitAbiN "k t b" FiCAL ["book"]
N------S2R kitAbi "k t b" FiCAL ["book"]
N------S2A kitAbi "k t b" FiCAL ["book"]
N------S2L kitAbi "k t b" FiCAL ["book"]
N------S4R kitAba "k t b" FiCAL ["book"]
N------S4A kitAba "k t b" FiCAL ["book"]
N------S4L kitAba "k t b" FiCAL ["book"]
N------P1I kuttAbuN "k t b" FACiL ["writer","author","clerk"]
N------P1R kuttAbu "k t b" FACiL ["writer","author","clerk"]
N------P1A kuttAbu "k t b" FACiL ["writer","author","clerk"]
N------P1L kuttAbu "k t b" FACiL ["writer","author","clerk"]
N------P2I kuttAbiN "k t b" FACiL ["writer","author","clerk"]
N------P2R kuttAbi "k t b" FACiL ["writer","author","clerk"]
N------P2A kuttAbi "k t b" FACiL ["writer","author","clerk"]
N------P2L kuttAbi "k t b" FACiL ["writer","author","clerk"]
N------P4R kuttAba "k t b" FACiL ["writer","author","clerk"]
N------P4A kuttAba "k t b" FACiL ["writer","author","clerk"]
N------P4L kuttAba "k t b" FACiL ["writer","author","clerk"]

ElixirFM> :q
```

Yet trees are not 'trees', until
so named and seen—
and never were so named, till
those had been

John Ronald Reuel Tolkien,
Mythopoeia

Chapter 7

MorphoTrees

The classical concept of morphological analysis is, technically, to take individual subparts of some linear representation of an utterance, such as orthographic words, interpret them regardless of their context, and produce for each of them a list of morphological readings revealing what hypothetical processes of inflection or derivation the given form could be a result of. One example of such a list is seen at the top of Figure 7.1.

The complication has been, at least with Arabic, that the output information can be rather involved, yet it is linear again while some explicit structuring of it might be preferable. The divergent analyses are not clustered together according to their common characteristics. It is very difficult for a human to interpret the analyses and to discriminate among them. For a machine, it is undefined how to compare the differences of the analyses, as there is no disparity measure other than unequalness.

MorphoTrees (Smrž and Pajas, 2004) is the idea of building effective and intuitive hierarchies over the information presented by morphological systems (Figure 7.1). It is especially interesting for Arabic and the Functional Arabic Morphology, yet, it is not limited to the language, nor to the formalism, and various extensions are imaginable.

7.1 The MorphoTrees Hierarchy

As an inspiration for the design of the hierarchies, consider the following analyses of the string `fhm` فهم. Some readings will interpret it as just one token related to the notion of 'understanding', but homonymous for several lexical units, each giving many inflected forms, distinct phonologically despite their identical spelling in the ordinary non-vocalized text. Other readings will decompose the string into two co-occurring tokens, the first one, in its non-vocalized form `f` ف, standing for an unambiguous conjunction, and the other one, `hm` هم, analyzed as a verb, noun, or pronoun, each again ambiguous in its functions.

Clearly, this type of concise and 'structured' description does not come ready-made—we have to construct it on top of the overall morphological knowledge. We can take the output solutions of morphological analyzers and process them according to our requirements on tokenization and 'functionality' stated above. Then, we can merge the analyses

Morphs	Form	Token Tag	Lemma	Morph-Oriented Gloss
|laY+(null)	*ʾālā*	VP-A-3MS--	*ʾālā*	promise/take an oath + he/it
|liy~	*ʾālīy*	A---------	*ʾālīy*	mechanical/automatic
|liy~+u	*ʾālīy-u*	A-------1R	*ʾālīy*	mechanical/automatic + [def.nom.]
|liy~+i	*ʾālīy-i*	A-------2R	*ʾālīy*	mechanical/automatic + [def.gen.]
|liy~+a	*ʾālīy-a*	A-------4R	*ʾālīy*	mechanical/automatic + [def.acc.]
|liy~+N	*ʾālīy-un*	A-------1I	*ʾālīy*	mechanical/automatic + [indef.nom.]
|liy~+K	*ʾālīy-in*	A-------2I	*ʾālīy*	mechanical/automatic + [indef.gen.]
|l +	*ʾāl*	N--------R	*ʾāl*	family/clan
+ iy	*-ī*	S----1-S2-	*ʾanā*	my
IilaY	*ʾilā*	P---------	*ʾilā*	to/towards
Iilay +	*ʾilay*	P---------	*ʾilā*	to/towards
+ ya	*-ya*	S----1-S2-	*ʾanā*	me
Oa+liy+(null)	*ʾa-lī*	VIIA-1-S--	*waliya*	I + follow/come after + [ind.]
Oa+liy+a	*ʾa-liy-a*	VISA-1-S--	*waliya*	I + follow/come after + [sub.]

Figure 7.1 Analyses of the orthographic word AlY الى turned into the MorphoTrees hierarchy. The full forms and morphological tags in the leaves are schematized to triangles.

and their elements into a five-level hierarchy similar to that of Figure 7.2. The leaves of it are the full forms of the tokens plus their tags as the atomic units. The root of the hierarchy represents the input string, or generally the input entity (some linear or structured subpart of the text). Rising from the leaves up to the root, there is the level of lemmas of the lexical units, the level of non-vocalized canonical forms of the tokens, and the level of decomposition of the entity into a sequence of such forms, which implies the number of tokens and their spelling.

Let us note that the MorphoTrees hierarchy itself might serve as a framework for evaluating morphological taggers, lemmatizers and stemmers of Arabic, since it allows for resolution of their performance on the different levels, which does matter with respect to the variety of applications.

7.2 MorphoTrees Disambiguation

The linguistic structures that get annotated as trees are commonly considered to belong to the domain of syntax. Thanks to the excellent design and programmability of TrEd,[1] the general-purpose tree editor written by Petr Pajas, we could happily implement an extra annotation mode for the disambiguation of MorphoTrees, too. We thus acquired a software environment integrating all the levels of description in PADT.

The annotation of MorphoTrees rests in selecting the applicable sequence of tokens that analyze the entity in the context of the discourse. In a naive setting, an annotator would be left to search the trees by sight, decoding the information for every possible analysis before coming across the right one. If not understood properly, the supplementary levels of the hierarchy would rather tend to be a nuisance . . .

Instead, MorphoTrees in TrEd take great advantage of the hierarchy and offer the option to restrict one's choice to subtrees and hide those leaves or branches that do not conform to the criteria of the annotation. Moreover, many restrictions are applied automatically, and the decisions about the tree can be controlled in a very rapid and elegant way.

The MorphoTrees of the entity `fhm` فهم in Figure 7.2 are in fact annotated already. The annotator was expecting, from the context, the reading involving a conjunction. By pressing the shortcut `c` at the root node, he restricted the tree accordingly, and the only one eligible leaf satisfying the `C---------` tag restriction was selected at that moment. Nonetheless, the *fa-* ف 'so' conjunction is part of a two-token entity, and some annotation of the second token must also be performed. Automatically, all inherited restrictions were removed from the `hm` هم subtree (notice the empty tag in the flag over it), and the subtree unfolded again. The annotator moved the node cursor[2] to the lemma for the pronoun, and restricted its readings to the nominative `--------1-` by pressing another mnemonic shortcut `1`, upon which the single conforming leaf *hum* هم 'they' was selected automatically. There were no more decisions to make and the annotation proceeded to the next entity of the discourse.

Alternatively, the annotation could be achieved merely by typing `s1`. The restrictions would unambiguously lead to the nominative pronoun, and then, without human intervention, to the other token, the unambiguous conjunction. These automatic decisions need no linguistic model, and yet they are very effective.

Incorporating restrictions or forking preferences sensitive to the surrounding annotations is in principle just as simple, but the concrete rules of interaction may not be easy to find. Morphosyntactic constraints on multi-token word formation are usually hard-wired inside analyzers and apply within an entity—still, certain restrictions might be generalized and imposed automatically even on the adjacent tokens of successive entities, for

[1] TrEd is open-source and is documented at `http://ufal.mff.cuni.cz/~pajas/tred/`.

[2] Navigating through the tree or selecting a solution is of course possible using the mouse, the cursor arrows, and the many customizable keyboard shortcuts. Restrictions are a convenient option to consider.

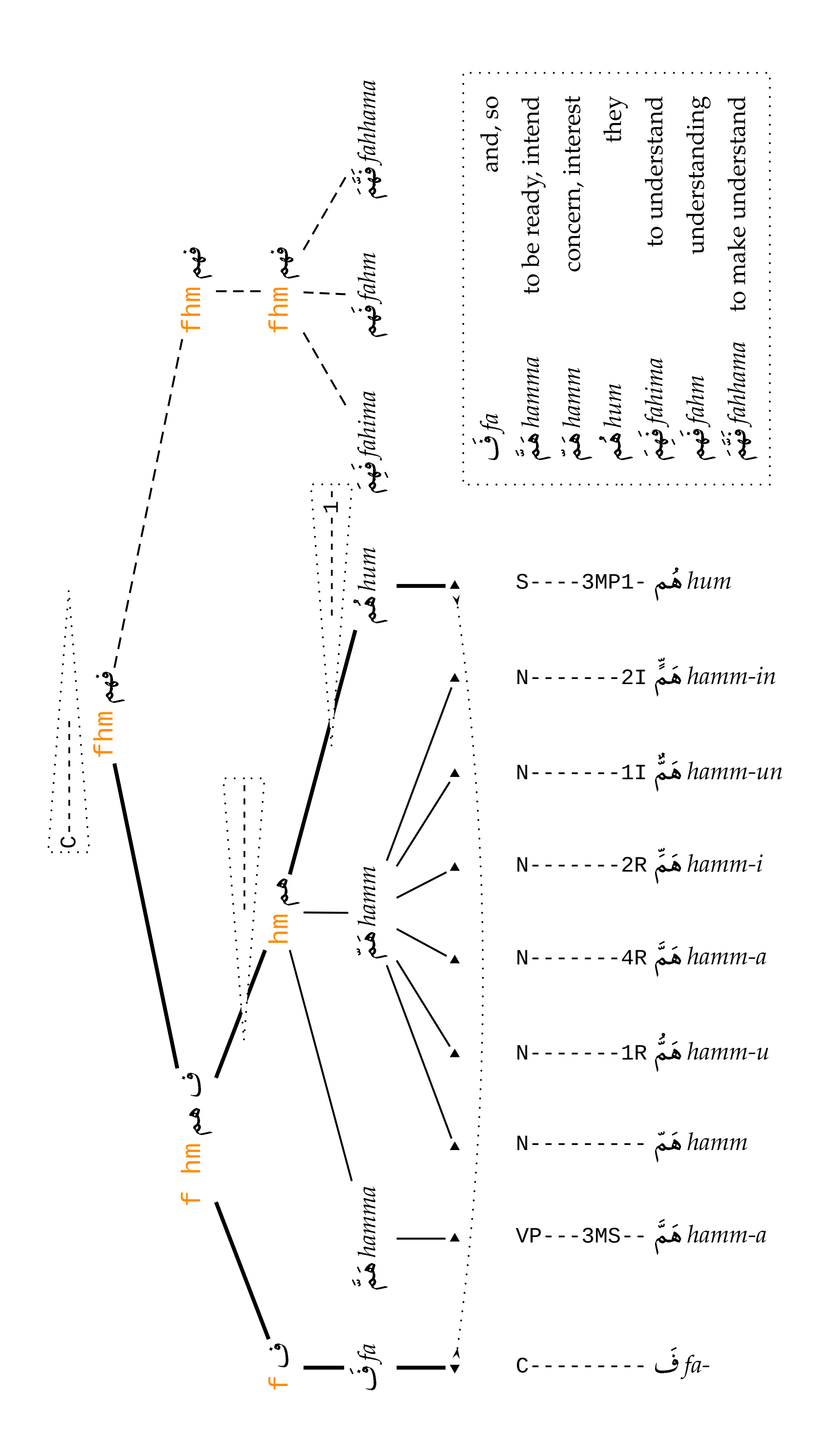


Figure 7.2 MorphoTrees of the orthographic string fhm فهم including annotation with restrictions. The dashed lines indicate there is no solution suiting the inherited restrictions in the given subtree. The dotted line symbolizes there might be implicit morphosyntactic constraints between the adjacent tokens in the analyses.

instance. Eventually, annotation of MorphoTrees might be assisted with real-time tagging predictions provided by some independent computational module.

Just for illustration, let us extract a snippet from the MorphoTrees implementation written in Perl. The function `restrict` applied to two tags `--------1-` and `S----3MP4-` would yield `S----3MP1-`. Naturally, the first argument can specify more positions at once, e.g. `VI-A--F---`. Nodes become hidden if their tags do not conform to the inherited restrictions, which we can formulate using `restrict` in the criterion below:

```
sub restrict {

    my @restrict = split //, length $_[0] == $dims ? $_[0] : '-' x $dims;
    my @inherit = split //, $_[1];

    return join '', map { $restrict[$_] eq '-' && defined $inherit[$_] ?
                          $inherit[$_] : $restrict[$_] } 0 .. $#restrict;
}

$node->{'hide'} = 'hide' if $node->{'tag'} ne
                            restrict($node->{'inherit'}, $node->{'tag'});
```

7.3 Further Discussion

Hierarchization of the selection task seems to be the most important contribution of the idea. The suggested meaning of the levels of the hierarchy mirrors the linguistic theory and also one particular strategy for decision-making, neither of which are universal. If we adapt MorphoTrees to other languages or hierarchies, the power of trees remains, though—efficient top-down search or bottom-up restrictions, gradual focusing on the solution, refinement, inheritance and sharing of information, etc.

The levels of MorphoTrees are extensible internally (More decision steps for some languages?) as well as externally in both directions (Analyzed entity becoming a tree of perhaphs discontiguous parts of a possible idiom? Leaves replaced with derivational trees organizing the morphs of the tokens?) and the concept incites new views on some issues encompassed by morphological analysis and disambiguation.

In PADT, whose MorphoTrees average roughly 8–10 leaves per entity depending on the data set while the result of annotation is 1.16–1.18 tokens per entity, restrictions as a means of direct access to the solutions improve the speed of annotation significantly.[3]

How would the first and the second level below the root in MorphoTrees be defined, if we used a different tokenization scheme? Some researchers do not reconstruct the canonical non-vocalized forms as we do, but only determine token boundaries between the characters of the original string (cf. Diab et al., 2004, Habash and Rambow, 2005). Our

[3]We have also implemented MorphoLists, a format of the data that simulates the classical non-hierarchical approach to disambiguation, with the MorphoTrees annotation context being reused for this format, too.

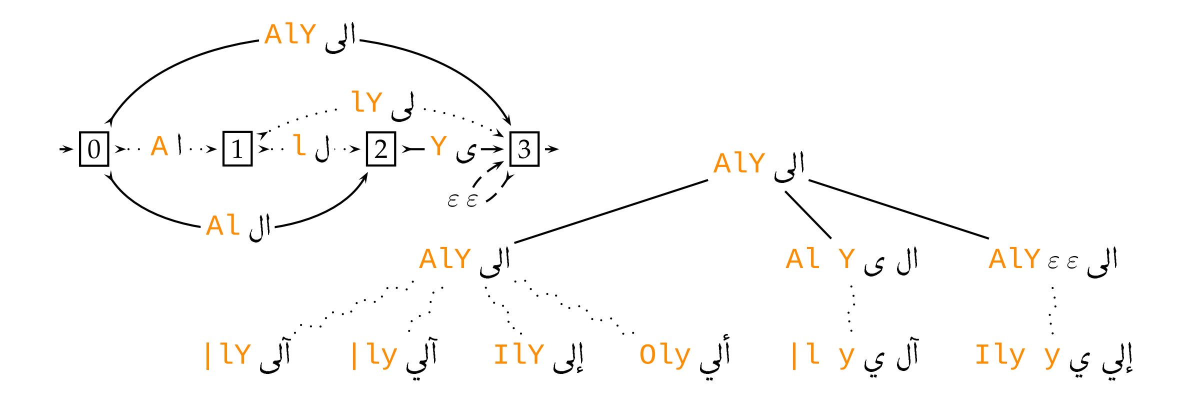


Figure 7.3 Discussion of partitioning and tokenization of orthographic strings.

point in doing the more difficult job is that (a) we are interested in such level of detail (b) disambiguation operations become more effective if the hierarchy reflects more distictions (i.e. decisions are specific about alternatives).

The relation between these tokenizations is illustrated in Figure 7.3. The graph on the left depicts the three 'sensible' ways of partitioning the input string AlY الى in the approach of (Diab et al., 2004), where characters are classified to be token-initial or not. In the graph, boundaries between individual characters are represented as the numbered nodes in the graph. Two of the valid tokenizations of the string are obtained by linking the boundaries from 0 to 3 following the solid edges in the directions of the arrows. The third partitioning AlY ε ε الى indicates that there is another fictitious boundary at the end of the string, yielding some 'empty word' ε ε, which together corresponds to leaping over the string at once and then taking the dashed edge in the graph.

Even though conceptually sound, this kind of partitioning may not be as powerful and flexible as what MorphoTrees propose, because it rests in classifying the input characters only, and not actually constructing the canonical forms of tokens as an *arbitrary function* of the input. Therefore, it cannot undo the effects of orthographic variation (Buckwalter, 2004b), nor express other useful distinctions, such as recover the spelling of *tāʾ marbūṭa* or normalize *hamza* carriers.

We can conclude with the tree structure of Figure 7.3. The boundary-based tokenizations are definitely not as detailed as those of MorphoTrees given in Figure 7.1, and might be occasionally thought of as another intermediate level in the hierarchy. But as they are not linguistically motivated, we do not establish the level as such.

In any case, we propose to evaluate tokenizations in terms of the Longest Common Subsequence (LCS) problem (Crochemore et al., 2000, Konz and McQueen, 2000–2006). The tokens that are the members of the LCS with some referential tokenization, are considered correctly recognized. Dividing the length of the LCS by the length of one of the sequences, we get recall, doing it for the other of the sequences, we get precision. The harmonic mean of both is $F_{\beta=1}$-measure (cf. e.g. Manning and Schütze, 1999).

Lexical closure means that an anonymous function carries its native environment wherever it goes, just like some tourists I have met.

Mark Jason Dominus, *Higher-Order Perl*

Chapter 8

Lexicon versus Treebank

Let us give a rough outline of the structure of linguistic description in the framework of Functional Generative Description and motivate our specific concerns about Arabic within the Prague Arabic Dependency Treebank.

8.1 Functional Description of Language

Prague Arabic Dependency Treebank (Hajič et al., 2004a,b) is a project of analyzing large amounts of linguistic data in Modern Written Arabic in terms of the formal representation of language that originates in the Functional Generative Description (Sgall et al., 1986, Sgall, 1967, Panevová, 1980, Hajičová and Sgall, 2003).

Within this theory, the formal representation delivers the linguistic meaning of what is expressed by the surface realization, i.e. the natural language. The description is designed to enable generating the natural language out of the formal representations. By constructing the treebank, we provide a resource for computational learning of the correspondences between both languages, the natural and the formal.

Functional Generative Description stresses the principal difference between the *form* and the *function* of a linguistic entity,[1] and defines the kinds of entities that become the building blocks of the respective level of linguistic description—be it underlying or surface syntax, morphemics, phonology or phonetics.

In this theory, a *morpheme* is the least unit representing some linguistic meaning, and is understood as a function of a *morph*, i.e. a composition of phonemes in speech or orthographic symbols in writing, which are in contrast the least units capable of distinguishing meanings.

Similarly, morphemes build up the units of syntactic description, and assume values of abstract categories on which the grammar can operate. In FGD, this very proposition

[1] It seems important to note that the assignment of function to form is arbitrary, i.e. subject to convention—while Kay (2004) would recall *l'arbitraire du signe* in this context, Hodges (2006, section 2) would draw a parallel to *waḍʿ* وَضع 'convention'.

implies a complex suite of concepts, introduced with their own terminology and constituting much of the theory. For our purposes here, though, we would only like to reserve the generic term *token* to denote a syntactic unit, and defer any necessary refinements of the definition to later sections.

The highest abstract level for the description of linguistic meaning in FGD is that of the underlying syntax. It comprises the means to capture all communicative aspects of language, including those affecting the form of an utterance as well as the information structure of the discourse. From this deep representation, one can generate the lower levels of linguistic analysis, in particular the surface syntactic structure of a sentence and its linear sequence of phonemes or graphemes.

In the series of Prague Dependency Treebanks (Hajič et al., 2001, 2006, Cuřín et al., 2004, Hajič et al., 2004a), this generative model of the linguistic process is inverse and annotations are built, with minor modifications to the theory, on the three layers denoted as morphological, analytical and tectogrammatical.

Morphological annotations identify the textual forms of a discourse lexically and recognize the morphosyntactic categories that the forms assume. Processing on the analytical level describes the superficial syntactic relations present in the discourse, whereas the tectogrammatical level reveals the underlying structures and restores the linguistic meaning (cf. Sgall et al., 2004, for what concrete steps that takes).

Linguistic data, i.e. mostly newswire texts in their original written form, are gradually analyzed in this system of levels, and their linguistic meaning is thus reconstructed and made explicit.

8.1.1 Analytical Syntax

The tokens with their disambiguated grammatical information enter the annotation of analytical syntax (Žabokrtský and Smrž, 2003, Hajič et al., 2004b).

This level is formalized into dependency trees the nodes of which are the tokens. Relations between nodes are classified with analytical syntactic functions. More precisely, it is the whole subtree of a dependent node that fulfills the particular syntactic function with respect to the governing node.

In Figures 8.1–8.4, we analyze the following sentence from our treebank:

(1) وفي ملف الأدب طرحت المجلة قضية اللغة العربية والأخطار التي تهددها.

Wa-fī milaffi ʾl-ʾadabi ṭaraḥati ʾl-mağallatu qaḍīyata ʾl-luġati ʾl-ʿarabīyati wa-ʾl-ʾaḫṭāri ʾllatī tuhaddiduhā.

‘In the section on literature, the magazine presented the issue of the Arabic language and the dangers that threaten it.’

Figure 8.1 depicts a simplified plain verbal sentence. In Figure 8.3, we extend the structure with coordination and a subordinate relative clause. Coordination is depicted with a diamond node and dashed 'dependency' edges between the coordination node and its member coordinants.

Both clauses and nominal expressions can assume the same analytical functions—the attributive clause in our example is Atr, just like in the case of nominal attributes. Pred denotes the main predicate, Sb is subject, Obj is object, Adv stands for adverbial. AuxP, AuxY and AuxK are auxiliary functions of specific kinds.

The coordination relation is different from the dependency relation, however, we can depict it in the tree-like manner, too. The coordinative node becomes Coord, and the subtrees that are the members of the coordination are marked as such (cf. dashed edges). Dependents modifying the coordination as a whole would attach directly to the Coord node, yet would not be marked as coordinants—therefrom, the need for distinguishing coordination and pure dependency in the trees.

The immediate-dominance relation that we capture in the annotation is independent of the linear ordering of words in an utterance, i.e. the linear-precedence relation (Debusmann et al., 2005, Debusmann, 2006). Thus, the expressiveness of the dependency grammar is stronger than that of phrase-structure context-free grammar. The dependency trees can become non-projective by featuring crossing dependencies, which reflects the possibility of relaxing word order while preserving the links of grammatical government.

8.1.2 Tectogrammatics

The analytical syntax is yet a precursor to the deep syntactic annotation (Sgall et al., 2004, Mikulová et al., 2006). We can note these characteristics of the tectogrammatical level, and compare the representations of example (1) in Figures 8.1 vs. 8.3 and Figures 8.2 vs. 8.4:

deleted nodes only autosemantic lexemes and coordinative nodes are involved in tectogrammatics; synsemantic lexemes, such as prepositions or particles, are deleted from the trees and may instead reflect in the values of deep grammatical categories, called *grammatemes*, that are associated with the relevant autosemantic nodes

inserted nodes autosemantic lexemes that do not appear explicitly in the surface syntax, yet that are demanded as obligatory by valency frames or by other criteria of tectogrammatical well-formedness, are inserted into the deep syntactic structures; the elided lexemes may be copies of other explicit nodes, or may be restored even as generic or unspecified

functors are the tectogrammatical functions describing deep dependency relations; the underlying theory distinguishes *arguments* (inner participants: ACTor, PATient, ADDRessee, ORIGin, EFFect) and *adjuncts* (free modifications, e.g.: LOCation, CAUSe,

MANNer, TimeWHEN, ReSTRictive, APPurtenance) and specifies the type of coordination (e.g. CONJunctive, DISJunctive, ADVerSative, ConSeQuential)

grammatemes are the deep grammatical features that are necessary for proper generation of the surface form of an utterance, given the tectogrammatical tree as well (cf. Hajič et al., 2004b)

coreference pronouns are matched with the lexical mentions they refer to; we distinguish *grammatical* coreference (the coreferent is determined by grammar) and *textual* coreference (otherwise); in Figure 8.4, the black dotted arcs indicate grammatical coreference, the loosely dotted red curves denote textual coreference.

Note the differences in the set of nodes actually represented, esp. the restored ADDRessee which is omitted in the surface form of the sentence, but is obligatory in the valency frame of the semantics of the PREDicate.

8.2 Dependency and Inherent vs. Inflectional Properties

Analytical syntax makes the agreement relations more obvious. We can often use those relations to infer information on inherent lexical properties as to gender, number, and humanness, as other words in the relation can, with their inflectional properties, provide enough constraints.

So this problem is a nice example for constraint programming. Our experiments with the treebank so far have been implemented in Perl, and the inference algorithm was not optimal. Neither was the handling of constraints that (perhaps by an error in the annotation) contradict the other ones. Anyway, we did arrive at promising preliminary results.

These experiments have not been fully completed, though, and their revision is needed. In view of that, we consider formulating the problem in the Mozart/Oz constraint-based programming environment (Van Roy and Haridi, 2004, chapters 9 and 12).

8.3 Tectogrammatics and Derivational Morphology

We can define default derivations of participles and deverbal nouns, the *maṣdar*s, or consider transformations of patterns between different derivational forms, like in the case of Czech where lexical-semantic shifts are also enforced in the valency theory (cf. Žabokrtský, 2005). If the default happens to be inappropriate, then a lexical entry can be extended to optionally include the lexicalized definition of the information that we might require.

The concrete transformations that should apply on the tectogrammatical level are a research in progress, performed by the whole PADT team.

The ability to do the transformations, however, is expected in near future as a direct extension of the ElixirFM system.

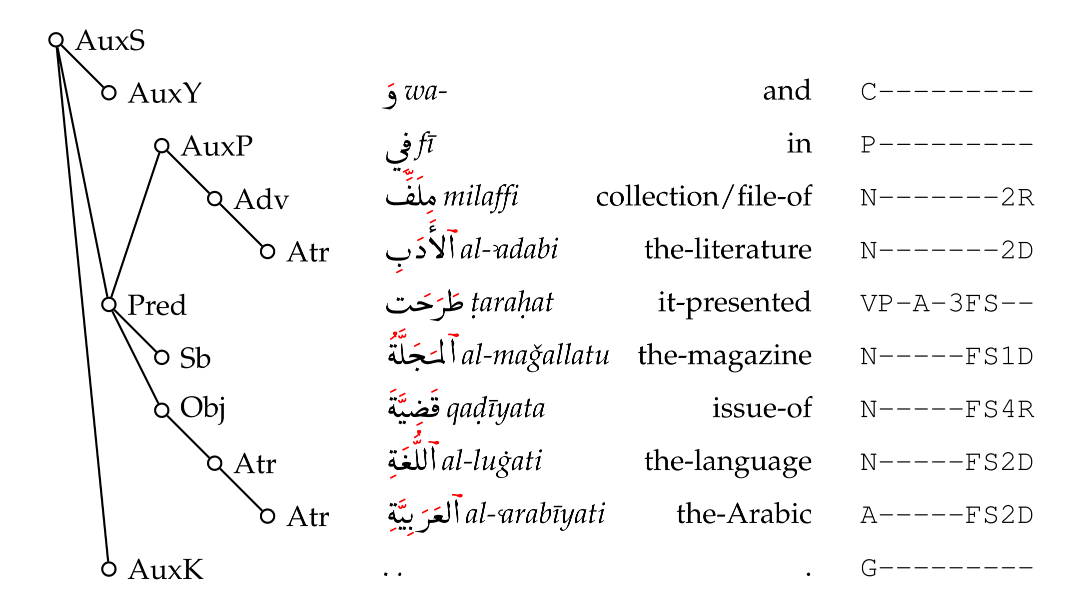


Figure 8.1 In the section on literature, the magazine presented the issue of the Arabic language. Analytical representation.

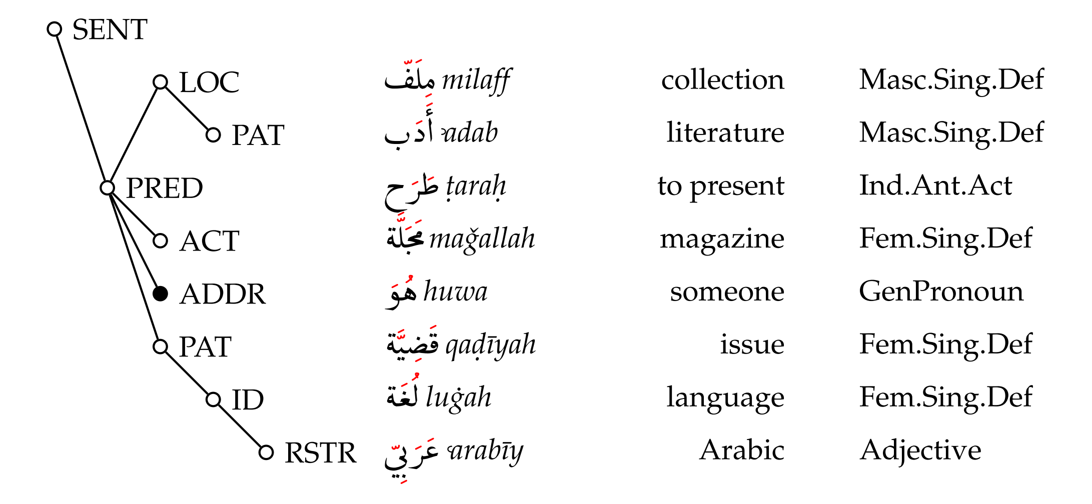


Figure 8.2 In the section on literature, the magazine presented the issue of the Arabic language. Tectogrammatics.

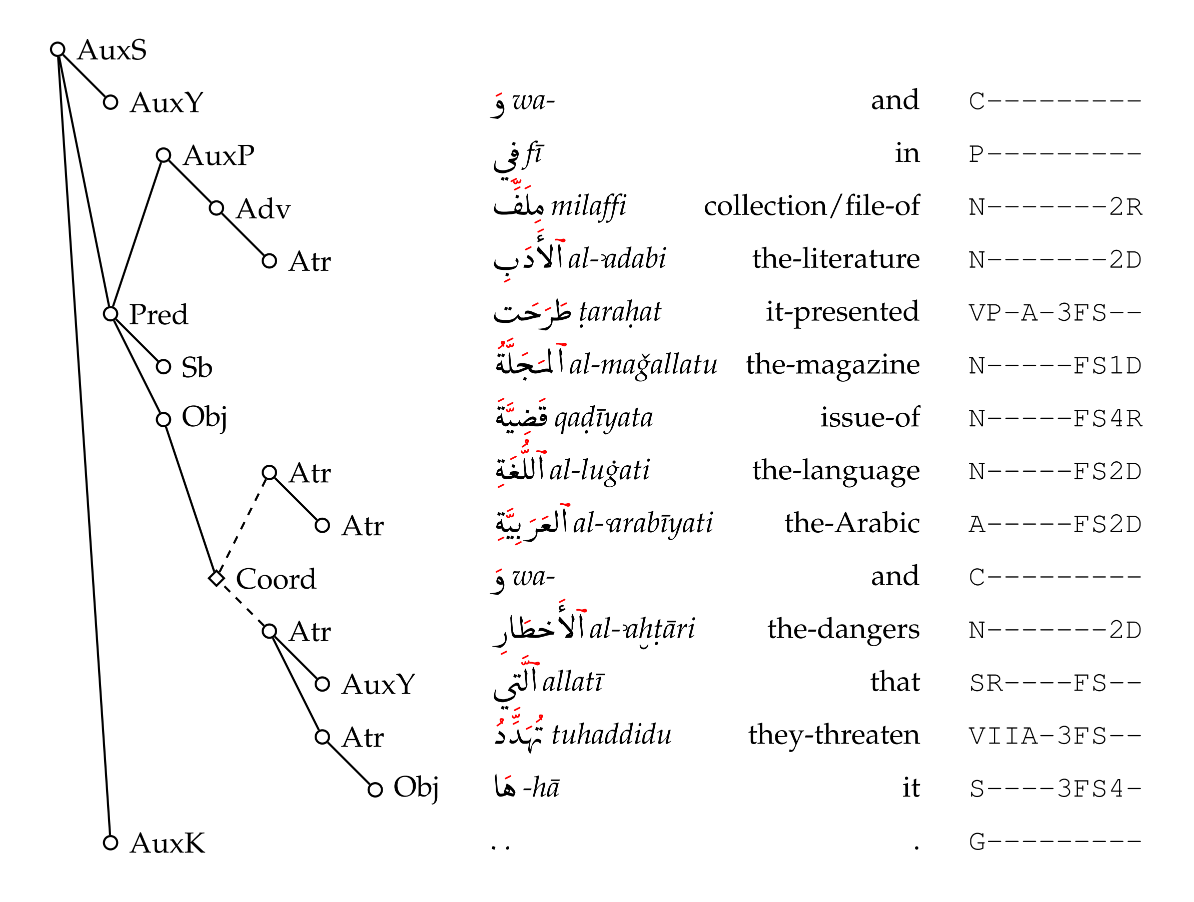


Figure 8.3 In the section on literature, the magazine presented the issue of the Arabic language and the dangers that threaten it. Analytical representation.

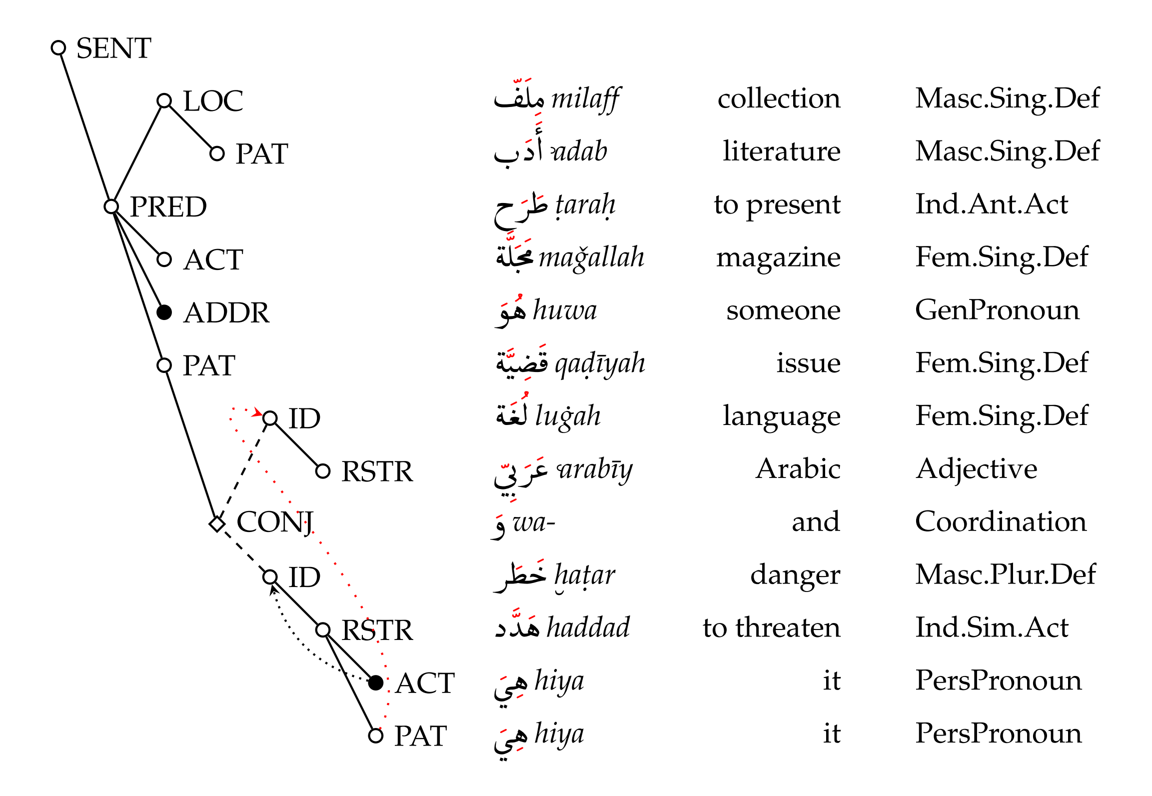


Figure 8.4 In the section on literature, the magazine presented the issue of the Arabic language and the dangers that threaten it. Tectogrammatics.

You'll be relieved to know that I have no opinion on it. :-)

Larry Wall quoted on *use Perl;*

Chapter 9

Encode Arabic

This chapter contains details about the implementations related to processing the ArabTEX notation and its extensions described in Chapter 2. The mentioned software is open-source and is available via `http://sourceforge.net/projects/encode-arabic/`.

9.1 Extending ArabTEX

The alocal package implements some of the notational extensions of Encode Arabic to work in ArabTEX. It is invoked by ArabTEX as the file containing local modifications.

The acolor package adds colorful typesetting to ArabTEX. I would like to thank Karel Mokrý for having implemented the core of this functionality, and for having introduced me to ArabTEX almost a decade ago.

There are further options for typesetting the Arabic script with TEX or XeTEX, cf. (Jabri, 2006) resp. (Charette, 2007). The latter system, ArabXeTEX, implements translation of the ArabTEX notation into Unicode via the TECkit conversion engine distributed with XeTEX.

9.2 Encode Arabic in Perl

The Perl implementation of Encode Arabic is documented at `http://search.cpan.org/dist/Encode-Arabic/`.

9.3 Encode Arabic in Haskell

We present parts of an implementation of a Haskell library for processing the Arabic language in the ArabTEX transliteration (Lagally, 2004), a non-trivial and multi-purpose notation for encoding Arabic orthographies and phonetic transcriptions in parallel. Our approach relies on the Pure Functional Parsing library developed in (Ljunglöf, 2002), which we accommodate to our problem and partly extend. We promote modular design in systems for modeling or processing natural languages.

9.3.1 Functional Parsing

Parsing is the process of recovering structured information from a linear sequence of symbols. The formulation of the problem in terms of functional programming is well-known, and both excellent literature and powerful computational tools are available (Wadler, 1985, 1995, Hutton and Meijer, 1996, 1998, Swierstra and Duponcheel, 1996, Swierstra, 2001, Leijen and Meijer, 2001, Marlow, 2001, 2003) .

The overall parsing process can be divided into layers of simpler parsing processes. Typically, there is one lexing/scanning phase involving deterministic finite-state techniques (Chakravarty, 1999), and one proper parsing phase resorting to context-free or even stronger computation to resolve the language supplied by the lexer.

Most of the libraries, however, implement parsers giving fixed data types for them, and implicitly restrict the parsing method to a single technique. Lexing with Chakravarty's CTK/Lexers and proper parsing with Leijen's Parsec would imply 'different' programming.

A unifying and theoretically instructive account of parsing in a functional setting was presented by Peter Ljunglöf in his licentiate thesis (Ljunglöf, 2002). Pure Functional Parsing, esp. the parts discussing recursive-descent parsing using parser combinators, seemed the right resource for us to implement the grammars we need. This library in Haskell, called PureFP in the sequel, abstracts away from the particular representation of the parser's data type. It provides a programming interface based on type classes and methods, leaving the user the freedom to supply the parser types and the processing functions quite independently of the descriptions in the grammars.

9.3.2 Encode Mapper

In the Encode Mapper module, we implement a lazy deterministic finite-state transducer. This kind of parser is in the first approximation represented as a trie, i.e. a tree structure built from the lexical specification in the grammar. In the trie, edges correspond to input symbols and nodes to states in which output results are possibly stored. A path from the root of the trie to a particular node encodes the sequence of symbols to be recognized in the input, if the result associated with that node is to be emitted.

Chakravarty (1999) gave an account on building tries with possible repetitions and cycles. The results can be actions or meta-actions—the latter being a device to escape to non-regular capabilities of the parsers, such as recognizing nested expressions or changing the parser dynamically during parsing. The parser does not allow ambiguous results, and parsing is controlled by the principle of the longest match.

Ljunglöf (2002) re-formulates such kind of parsing in terms of his library, and offers further explanation and discussion on the whole issue. While he develops several data representations of tries suited for ambiguous grammars and supporting efficient sharing of subtries in memory, he leaves the question of longest match aside.

The Encode Mapper implements something in between the two. The nature of our `Mapper` parser is the `AmbExTrie` parser described in detail in (Ljunglöf, 2002, sec. 4.3). We add to it the abilities to 'cheat' by rewriting the input with other symbols while still producing results of the general type `a`, and to parse ambiguously using a breadth-first or a depth-first longest match algorithm.

```
module Encode.Mapper where

import PureFP.OrdMap
import PureFP.Parsers.Parser

data Mapper s a = [Quit s a] :&: Map s (Mapper s a)
                | forall b . FMap (b -> a) (Mapper s b)
type Quit s a = ([s], a)
```

A node in the `Mapper s a` trie is the tuple `:&:` of a list of results of type `Quit s a` and a finite map from input symbols to subtries, mimicking the labeled directed edges. For better memory representation, subtries can be wrapped into the `FMap` constructor introduced in the original work. For finite maps, we use Ljunglöf's `PureFP.OrdMap` similar to `Data.Map` of the Haskell libraries.

The `Quit s a` data type is a tuple of a sequence of input symbols to be returned to the input upon a match, and of the result to be reported.

Our notation for expressing grammars will use four new infix operators, the definition of which follows. The `|+|` appends an alternative rule. A completely matching input sequence with no cheating is combined with the result by `|.|`. In case of cheating, i.e. input rewriting, the matching and the cheating sequences are joined with `|-|`, and that is combined with the result via `|:|`.

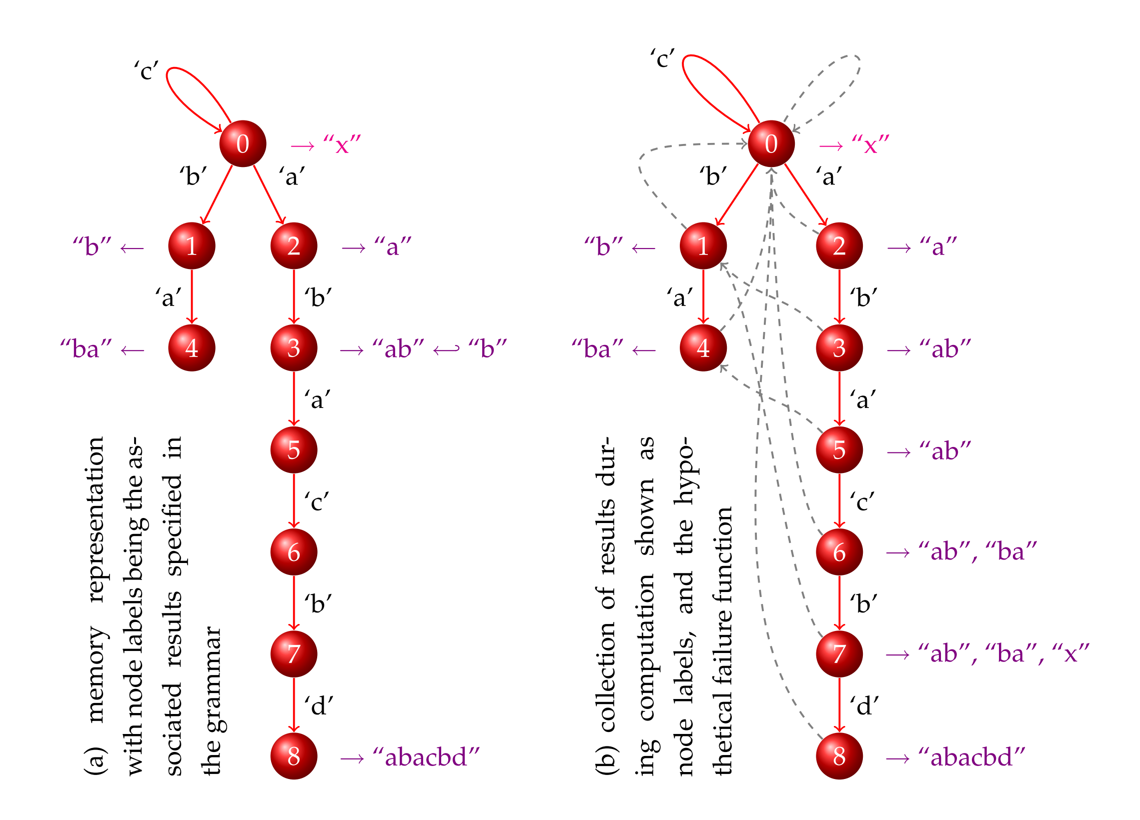
```
infix  4 |-|             -- rule =      "a"          |.| 1
infix  3 |:|, |.|        --        |+|  "a"          |.| 2
infixl 2 |+|             --        |+|  "b" |-| "aa" |:| 3

(|:|) :: InputSymbol s => (a -> Mapper s a) -> a
                                          -> Mapper s a
(|:|) x y = x y

(|-|) :: InputSymbol s => [s] -> [s] -> a -> Mapper s a
(|-|) x y z = syms x >> [returnQuit y z] :&: emptyMap

(|.|) :: InputSymbol s => [s] -> a -> Mapper s a
(|.|) x y = x |-| [] |:| y

(|+|) :: InputSymbol s => Mapper s a -> Mapper s a
                                          -> Mapper s a
(|+|) = (<+>)
```



(a) memory representation with node labels being the associated results specified in the grammar

(b) collection of results during computation shown as node labels, and the hypothetical failure function

Figure 9.1 Trie structures illustrating efficient longest match parsing.

The combinators in a grammar for `Mapper` are actually constructors that take care of building the trie gradually with rules. The trick to improve subtrie sharing rests in delaying function application on the results, and instead storing the modification functions inside the `FMap` value. We once again refer the reader to (Ljunglöf, 2002) for the proper discussion of this technique.

9.3.3 Longest Match Insight

Consider the following rules defining our example trie in Figure 9.1a:

```
trie :: Mapper Char [Char]
trie = (many (syms "c") >> return "x")   -- loop over "c"
        |+| "b"          |.| "b"      -- equal to syms "b"
        |+| "a"          |.| "a"
        |+| "ab" |-| "b" |:| "ab"     -- cheating with "b"
        |+| "ba"         |.| "ba"
        |+| "abacbd"     |.| "abacbd"
```

We can view this trie as a dictionary specifying a language. Its words are composed of the labels of those edges that create a path from the root to some node with a non-empty

list of results. Given an arbitrary input string, we would like to use this trie for finding and translating the longest non-overlapping substrings that belong to the dictionary. Yet, cheating and ambiguities will be allowed.

The inspiration for us is the Aho–Corasick algorithm for text search (Aho and Corasick, 1975). The important insight is the idea of a failure function, depicted with dashed lines in Figure 9.1b.

If parsing cannot proceed along the labeled, solid edges in the trie, there is no chance for any longer match. Then, we can report the longest match results we had collected (the node labels in this subfigure). But we also have to keep in mind that the input symbols accepted after the latest match can form a prefix of a new successful match. The failure function serves to move us to this prefix. From that point and with the same input symbol, we iterate this process until we succeed or reach the root in the trie. Then we parse the next input symbol, or conclude by following the failure functions and reporting the collected results until we reach the root.

In our implementation in Haskell, we are not going to construct the failure function in the trie. It would require to traverse the whole data structure before parsing a single symbol. Thus, we would lose the great advantage of lazy construction of the trie. The `Mapper` would also become restricted to finite tries only, which we cannot easily guarantee given the power of the combinator grammar.

Therefore, the parsing process itself will have to simulate what the failure function provides. We can either develop parallel hypotheses about where the actual match occurs (breadth-first search), or try the hypotheses in a backtracking manner using an accumulator for the non-matched input (depth-first search).

Let us see what results we get at this moment for our example `trie`:

```
ex :: [Char] -> [[[[Char]]]]
ex = unParseWide trie . parseWide trie [initPW trie id]

ex "ab"         →  [[["ab"],["b"]]]
ex "aba"        →  [[["ab","ba"]]]
ex "abacba"     →  [[["ab","ba","x","ba"]]]
ex "abacbc"     →  [[["ab","ba","x","b","x"]]]
ex "abacbccc"   →  [[["ab","ba","x","b","x"]]]
ex "eabacbee"   →  [[["x","ab","ba","x","b","x","x"]]]
```

In case of `trie′` with ‘problematic’ rules and `ex′` defined likewise:

```
trie′ = trie |+| "ab" |.| "ab"                -- ambiguity
             |+| ""   |-| "a"    |:| "y"      -- undefined
             |+| "c"  |-| "abac" |:| ""       -- expansion

ex′ "abab" → [[["ab","ba","b"]],   -- cheating difference
              [["ab","ab"],["b"]],
```

```
                  [["ab","ab"]]]
ex' "cc"    →  [[["x"]]]            -- unique longer match
ex' "c"     →  [[["x"]],
                  [[""],["ab","ba","x"]],
                  [[""],["ab","ba",""],["ab","ba","x"]], ...
                      -- iterating ["ab","ba",""] infinitely
ex' "cbd"   →  [[["x","b","x"]],
                  [["x","b","y"]],    -- double match for 'd'
                  [["","abacbd"]]]    -- finite 'c' rewriting
```

9.3.4 Encode Extend

In this section, we will describe the Encode Extend module implementing a general recursive-descent parser derived in the standard approach as a state transformer monad (Wadler, 1995). The 'extension' is that we decompose the state into the input being processed and the environment supplying any other needed parameters.

The `Extend e` parser is based on the `Standard` parser discussed in (Ljunglöf, 2002, sec. 3.2). Its state is a combination `InE s e` of a list of input symbols `s` and a stack of environment settings `e s`.

9.3.5 Encode Arabic

Before applying Encode Mapper and Encode Extend to the notation of ArabTEX, let us reformulate the idea of converting textual data from one encoding scheme into another in the way inspired by the Encode module in Perl (Kogai, 2002–2006).

We introduce the internal representation `UPoint` as the intermediate data type for these conversions. The distinction between this representation and characters **`Char`** is intentional, as 'decoded' and 'encoded' data are different entities. Since `UPoint` is an instance of the **`Enum`** class, the type's constructor and selector functions are available as **`toEnum`** and **`fromEnum`**, respectively.

```
module Encode where

newtype UPoint = UPoint Int deriving (Eq, Ord, Show)

instance Enum UPoint where
    fromEnum (UPoint x) = x
    toEnum = UPoint

class Encoding e where
    encode :: e -> [UPoint] -> [Char]
    decode :: e -> [Char] -> [UPoint]

    encode _ = map (toEnum . fromEnum)
    decode _ = map (toEnum . fromEnum)
```

Encoding schemes are modeled as data types `e` of the `Encoding` class, which defines the two essential methods. Developing a new encoding means to write a new module with a structure similar to `Encode.Arabic.Buckwalter` or `Encode.Unicode`, for instance.

```
module Encode.Arabic.Buckwalter (Buckwalter (..)) where

import Encode
import PureFP.OrdMap

data Buckwalter = Buckwalter | Tim deriving (Enum, Show)

instance Encoding Buckwalter where
    encode _ = recode (recoder decoded encoded)
    decode _ = recode (recoder encoded decoded)

recode :: (Eq a, Enum a, Enum b, Ord a)
       => Map a b -> [a] -> [b]
recode xry xs = [ lookupWith ((toEnum . fromEnum) x)
                             xry x | x <- xs ]

recoder :: Ord a => [a] -> [b] -> Map a b
recoder xs ys = makeMapWith const (zip xs ys)

decoded :: [UPoint]
decoded = map toEnum $ []

    ++ [0x0640] ++ [0x0623, 0x0624, 0x0625]
    ++ [0x060C, 0x061B, 0x061F]
    ++ [0x0621, 0x0622] ++ [0x0626 .. 0x063A]
    ++ [0x0641 .. 0x064A]
    ++ [0x067E, 0x0686, 0x06A4, 0x06AF]
    ++ [0x0660 .. 0x0669]
    ++ [0x0671] ++ [0x0651]
    ++ [0x064B .. 0x0650] ++ [0x0670] ++ [0x0652]

encoded :: [Char]
encoded = map id $ []

    ++ "_" ++ "OWI"
    ++ ",; ...? "
    ++ "'|" ++ "}AbptvjHxd*rzs$SDTZEg"
    ++ "fqklmnhwYy"
    ++ "PJVG"
    ++ ['0' .. '9']
    ++ "{" ++ "~"
    ++ "FNKaui" ++ "`" ++ "o"
```

The Buckwalter encoding is a lossless romanization of the standard Arabic script, and is a one-to-one mapping between the Unicode code points for Arabic and lower ASCII.

```
module Encode.Unicode (Unicode (..)) where

import Encode

data Unicode = Unicode | UCS deriving (Enum, Show)

instance Encoding Unicode
```

9.3.6 ArabTEX Encoding Concept

The ArabTEX typesetting system (Lagally, 2004) defines its own Arabic script meta-encoding that covers both contemporary and historical orthography in an excellent way. Moreover, the notation is human-readable as well as very natural to learn to write with. The notation itself is quite close to the phonetic transcription, yet extra features are introduced to make the conversion to script/transcription unambiguous.

Unlike other transliteration concepts based on the one-to-one mapping of graphemes, ArabTEX interprets the input characters in context to get their right meaning. Finding the glyphs of letters (initial, medial, final, isolated) and their ligatures is not the issue of encoding, but of visualizing only. Nonetheless, definite article assimilation, inference of *hamza* carriers and silent *ʾalif*s, treatment of auxiliary vowels, optional quoting of diacritics or capitalization, resolution of notational variants, and mode-dependent processing are the challenges for our parsing exercise now.

ArabTEX's implementation is documented in (Lagally, 1992), but the parsing algorithm for the notation has not been published. The TEX code of it is organized into deterministic-parsing macros, yet the complexity of the whole system makes consistent modifications or extensions by other users very difficult, if not impossible.

We are going to describe our own implementation of the interpreter, i.e. we will show how to `decode` the notation. To `encode` the Arabic script or its phonetic transcription into the ArabTEX notation requires some heuristics, if we want to achieve linguistically appropriate results. We leave these for future work.

9.3.7 Encode Arabic ArabTEX

This module relates the ArabTEX notation and the Arabic orthography. It provides definitions of the 'lexicon' of type `LowerUp`, which lists the distinct items in the notation and associates them with the information for their translation. Lexical items are identified in the lexing phase by an instance of Encode Mapper of type `Mapping`. The proper parsing phase uses Encode Extend parsers of type `Parsing`.

```
module Encode.Arabic.ArabTeX (ArabTeX (..)) where

import Encode
```

```
import Encode.Mapper
import Encode.Extend
import PureFP.OrdMap

data ArabTeX = ArabTeX | TeX deriving (Enum, Show)

instance Encoding ArabTeX where
  encode _ = error "'encode' is not implemented"
  decode _ = concat . parseFull decoderParsing .
             concat . parseLongest decoderMapping

type Parsing = Extend Env [Char] ([UPoint] -> [UPoint])
type Environ = Env [Char]
type Mapping = Mapper Char [[Char]]
type LowerUp = Map [Char] [UPoint]
```

The environment `Environ` is needed to store information bound to the context—otherwise, parsing rules would become complicated and inefficient.

```
data Mode = Nodiacritics | Novocalize | Vocalize | Fullvocalize

    deriving (Eq, Ord)

data Env i = Env { envQuote :: Bool, envMode :: Mode,
                   envWasla :: Bool, envVerb :: Bool,
                   envEarly :: [i] }

setQuote q (Env _ m w v e) = Env q m w v e
setMode  m (Env q _ w v e) = Env q m w v e
setWasla w (Env q m _ v e) = Env q m w v e
setVerb  v (Env q m w _ e) = Env q m w v e
setEarly e (Env q m w v _) = Env q m w v e

instance ExtEnv Env where

    initEnv = Env False Vocalize False False []
```

Note that the `decode` method ignores the encoding parameter. If our definitions were slightly extended, the `ArabTeX` data type could be parametrized with `Env` to allow user's own setting of the initial parsing environment, passed to `Encode.Extend.parseFull'`.

Lexicon The design of the lexicon cannot be simpler—the presented lexicon is nearly complete, but can be easily modified and extended. Lexical items are referred to in the mapping and the parsing phases by the sets they belong to, thus robustness is achieved.

```
define :: [([Char], [Int])] -> LowerUp
define l = makeMapWith const [ (x, map toEnum y) |
                               (x, y) <- l ]

consonant :: LowerUp
consonant = unionMap [sunny, moony, bound]

sunny = define [
    ( "t",  [ 0x062A ] ),    ( "^s", [ 0x0634 ] ),
    ( "_t", [ 0x062B ] ),    ( ".s", [ 0x0635 ] ),
    ( "d",  [ 0x062F ] ),    ( ".d", [ 0x0636 ] ),
    ( "_d", [ 0x0630 ] ),    ( ".t", [ 0x0637 ] ),
    ( "r",  [ 0x0631 ] ),    ( ".z", [ 0x0638 ] ),
    ( "z",  [ 0x0632 ] ),    ( "l",  [ 0x0644 ] ),
    ( "s",  [ 0x0633 ] ),    ( "n",  [ 0x0646 ] ) ]

invis = define [ ( "|",                [        ] ) ]

empty = define [ ( "",                 [ 0x0627 ] ) ]

shadda = define [ ( "*",               [ 0x0651 ] ) ]

silent = define [
    ( "A",  [ 0x0627 ] ),    ( "W",  [ 0x0627 ] ) ]

wasla = define [ ( "W",                [ 0x0671 ] ) ]

taaaa = define [
    ( "T",  [ 0x0629 ] ),    ( "H",  [ 0x0629 ] ) ]

bound = define [
    ( "'A", [ 0x0622 ] ),    ( "'w", [ 0x0624 ] ),
    ( "'a", [ 0x0623 ] ),    ( "'y", [ 0x0626 ] ),
    ( "'i", [ 0x0625 ] ),    ( "'|", [ 0x0621 ] ) ]

moony = define [
    ( "'",  [ 0x0621 ] ),    ( "f",  [ 0x0641 ] ),
    ( "b",  [ 0x0628 ] ),    ( "q",  [ 0x0642 ] ),
    ( "^g", [ 0x062C ] ),    ( "k",  [ 0x0643 ] ),
    ( ".h", [ 0x062D ] ),    ( "m",  [ 0x0645 ] ),
    ( "_h", [ 0x062E ] ),    ( "h",  [ 0x0647 ] ),
    ( "`",  [ 0x0639 ] ),    ( "w",  [ 0x0648 ] ),
    ( ".g", [ 0x063A ] ),    ( "y",  [ 0x064A ] ),

    ( "B",  [ 0x0640 ] ),    ( "c",  [ 0x0681 ] ),
                             ( "^c", [ 0x0686 ] ),
    ( "p",  [ 0x067E ] ),    ( ",c", [ 0x0685 ] ),
    ( "v",  [ 0x06A4 ] ),    ( "^n", [ 0x06AD ] ),
    ( "g",  [ 0x06AF ] ),    ( "^l", [ 0x06B5 ] ),
```

```
        ( "^z", [ 0x0698 ] ),    ( ".r", [ 0x0695 ] ) ]

    vowel = define [
        ( "a",  [ 0x064E ] ),    ( "_a", [ 0x0670 ] ),
        ( "i",  [ 0x0650 ] ),    ( "_i", [ 0x0656 ] ),
        ( "u",  [ 0x064F ] ),    ( "_u", [ 0x0657 ] ),
        ( "e",  [ 0x0650 ] ),    ( "o",  [ 0x064F ] ) ]

    multi = define [ ( "A",      [ 0x064E, 0x0627 ] ),
                     ( "I",      [ 0x0650, 0x064A ] ),
                     ( "U",      [ 0x064F, 0x0648 ] ),
                     ( "Y",      [ 0x064E, 0x0649 ] ),
                     ( "E",      [ 0x0650, 0x064A ] ),
                     ( "O",      [ 0x064F, 0x0648 ] ),
                     ( "_I",     [ 0x0650, 0x0627 ] ),
                     ( "_U",     [ 0x064F, 0x0648 ] ),
                     ( "aNY",    [ 0x064B, 0x0649 ] ),
                     ( "aNA",    [ 0x064B, 0x0627 ] ) ]

    nuuns = define [    ( "aN",          [ 0x064B ] ),
                        ( "iN",          [ 0x064D ] ),
                        ( "uN",          [ 0x064C ] ) ]

    other = define [ ( "_aY",    [ 0x0670, 0x0649 ] ),
                     ( "_aU",    [ 0x0670, 0x0648 ] ),
                     ( "_aI",    [ 0x0670, 0x064A ] ),

                ( "^A",   [ 0x064F, 0x0627, 0x0653 ] ),
                ( "^I",   [ 0x0650, 0x064A, 0x0653 ] ),
                ( "^U",   [ 0x064F, 0x0648, 0x0653 ] ) ]
```

Mapping The Encode Mapper tokenizes the input string into substrings that are items of the lexicon, or, which is very important, rewrites and normalizes the notation in order to make proper parsing clearer. It guarantees the longest match, no matter what order the rules or the lexicon are specified in.

```
    decoderMapping :: Mapper Char [[Char]]
    decoderMapping = defineMapping
        ( pairs [ sunny, moony, invis, empty, taaaa, silent,
                  vowel, multi, nuuns, other, sukun, shadda,
                  digit, punct, white ] )
        <+>  rules
        <+>  "" |.| error "Illegal symbol"

    rules :: Mapping
    rules =  "aN_A"  |-|  "aNY"   |:|     []          |+|
             "_A"    |-|  "Y"     |:|     []
```

```
    |+|  ruleVerbalSilentAlif |+|  ruleInternalTaaaa
    |+|  ruleLiWithArticle    |+|  ruleDefArticle
    |+|  ruleIndefArticle
    |+|  ruleMultiVowel       |+|  ruleHyphenedVowel
    |+|  ruleWhitePlusControl |+|  ruleIgnoreCapControl
    |+|  ruleControlSequence  |+|  rulePunctuation
```

In the rules, for instance, care of the silent *ʾalif* after the ending `aN` is taken, or the variants of definite article notation are unified. So is the notation for long vowels, which offers freedom to the user, yet must strictly conform to the context of the next syllable in orthography.

```
ruleIndefArticle =
    anyof [ c ++ m ++ "aNY"  |-| m ++ "aNY"  |:| [c]  |+|
            c ++ m ++ "aNA"  |-| m ++ "aNA"  |:| [c]  |+|
            c ++ m ++ "aN"   |-| m ++ "aNA"  |:| [c]
          | c <- elems [sunny, moony],
            m <- ["", "-", "\"", "-\""] ]
    |+| anyof [
            v ++ "''" ++ m ++ "aN" |-|
                         m ++ "aN" |:| [v, "'", "'"]  |+|
            v ++ "'"  ++ m ++ "aN" |-|
                         m ++ "aN" |:| [v, "'"]
          | v <- ["A", "a"], m <- ["", "-", "\"", "-\""]]

ruleDefArticle =
    anyof [     "l" ++ "-" ++ c ++ c         |-|
                       "-" ++ c              |:| [c]
            |    c <- elems [sunny, moony] ]

ruleMultiVowel =
    "iy"    |-|     "I"     |:|     []      |+|
    "Iy"    |-|     "yy"    |:|     ["i"]   |+|
    "uw"    |-|     "U"     |:|     []      |+|
    "Uw"    |-|     "ww"    |:|     ["u"]   |+|
    "aa"    |-|     "A"     |:|     []
    |+| anyof [
            "iy" ++ v  |-|  "y" ++ v      |:| ["i"]   |+|
            "uw" ++ v  |-|  "w" ++ v      |:| ["u"]
        | v <- elems [vowel, multi, nuuns, other] ++
             quote [vowel, multi, nuuns, other, sukun] ]

ruleControlSequence =
    do  x <- sym '\\' <:>
             some (anySymbol (['A'..'Z'] ++ ['a'..'z']))
        many whites
        return [x]
```

The list comprehension syntax in Haskell allows us to write rules in form of templates, where we can iterate over the elements of the lexical sets and give them symbolic names. In the last example of a `Mapping` rule, we combine consonants `c` and short vowels `v`, the latter possibly preceded by a quote `\"`.

```
ruleLiWithArticle =
    anyof [ "l" ++ v ++ "-a" ++ c ++ "-" ++ c   |-|
            "l" ++ v ++          c ++ "-" ++ c   |:| []
    |  c <- elems [sunny, moony], c /= "l",
       v <- elems [vowel, sukun] ++ quote [vowel, sukun]]
    |+| anyof [
        "l" ++ v ++ "-a" ++ c ++ "-" ++ c       |-|
        "l" ++ v ++ "|-" ++ c ++ c              |:| [] |+|
        "l" ++ v ++ "-a" ++ c ++ "-" ++ c ++ c |-|
        "l" ++ v ++ "|-" ++ c ++ c              |:| [] |+|
        "l" ++ v ++ "-a" ++ c ++ "-"            |-|
        "l" ++ v ++          c ++ "-"            |:| [] |+|
        "l" ++ v ++ "-a" ++ c ++ c              |-|
        "l" ++ v ++ "|-" ++ c ++ c              |:| []
    |  c <- elems [sunny, moony], c == "l",
       v <- elems [vowel, sukun] ++ quote [vowel, sukun]]
```

This rule alleviates a limitation in the original ArabTEX's coding of the prefixed words `li` and `la` when followed by a definite article. Due to an exceptional convention in orthography (cf. Lagally, 2004, sec. 4.1), *li-'l-mawzi* لالموز is not acceptable, and one has to write *lil-mawzi* للموز instead. Further complication comes with `l`, so *lil-lawzi* لللوز has to be transformed to *li-llawzi* للوز.

With the `ruleLiWithArticle` rewriting, one need not distinguish these anymore, and can just join words, like in other cases.

Parsing The result of complete parsing is `[UPoint]`. However, to avoid inefficient list concatenations, the simpler parsers being combined produce 'show' functions of type `([UPoint] -> [UPoint])`, composed sequentially with `plus`.

```
decoderParsing :: Extend Env [Char] [UPoint]
decoderParsing = (fmap (foldr ($) []) . again) $
            parseHyphen         <|>     parseHamza
    <|>     parseDefArticle
    <|>     parseDoubleCons     <|>     parseSingleCons
    <|>     parseInitVowel
    <|>     parseWhite          <|>     parsePunct
    <|>     parseDigit
    <|>     parseQuote          <|>     parseControl

infixr 7 `plus`                         -- infixr 9 .
                                        -- infixr 5 ++
```

```
plus :: (a -> b) -> (c -> a) -> c -> b
plus = (.)
```

Unlike `decoderMapping`, ordering of rules in `decoderParsing` does matter. The `again` and `<|>` combinators try the parsers in order, and if one succeeds, they continue again from the very first parser.

```
parseHyphen =        do  lower ["-"] []
                         resetEnv setEarly []
                         parseNothing
```

This one is rather simple. The `lower` parser consumes tokens that are specified in its first argument, and returns to the input the tokens of its second argument. Thus, `-` is removed from the input, the memory of previous input tokens is erased with `setEarly`, and no new output is produced, i.e. `parseNothing = return id`.

Parsing an assimilated definite article is perhaps even more intuitive, once `ruleDefArticle` is in effect. We look for any consonant `c` followed by a hyphen `-` and the same consonant. If we succeed, we return the two consonants back to the input, as they will form a regular 'syllable' eventually. We look up the translation for the letter `l` in the `sunny` set, and make it the output of the parser.

```
parseDefArticle =    do  c <- oneof [consonant]
                         lower ["-", c] [c, c]
                         upper ["l"] [sunny]
```

The compilation of 'syllables' in the Arabic script rests in putting vocalization marks onto the 'consonantal' letters. Processing of these marks is subject to the settings of the environment, in particular the `envQuote` and `envMode` values. We generalize `upper` to `upperWith`, to allow this processing.

```
parseDoubleCons =
  do  c <- oneof [consonant, taaaa, invis, silent]
      lower [c] []
      x <- upper [c] [consonant, taaaa, invis, silent]
      y <- upperWith shaddaControl
                 ["*"] [shadda]
      parseSyllVowel c (x `plus` y)

parseSyllVowel :: [Char] -> ([UPoint] -> [UPoint])
                                                -> Parsing
parseSyllVowel c x =
  do  v <- parseQuote <|> parseNothing >>
           oneof [vowel, multi, nuuns, other] <|>
           return ""
```

```
        y <- upperWith (vowelControl c)
                   [v] [vowel, multi, nuuns, other, sukun]
        completeSyllable [c, v] (x `plus` y)

completeSyllable :: [[Char]] -> ([UPoint] -> [UPoint])
                                             -> Parsing
completeSyllable l u =  do  resetEnv setQuote False
                            resetEnv setWasla True
                            resetEnv setEarly (reverse l)
                            return u
```

The definitions of `vowelControl` and `shaddaControl` are clear-cut. The `parseSingleCons` and `parseInitVowel` parsers go in the spirit of their namesakes that we have seen. The `parseControl` parser interprets control sequences that affect the parsing environment, including possible localization/nesting of the settings.

The last non-trivial parser is `parseHamza`. It does not produce any output, but computes the so-called carrier for the *hamza* consonant. In the `\setverb` mode, this carrier appears in the input after `''` or `'-'` or `'`. In the complementary `\setarab` mode, this carrier must be determined according to some rather complex orthographic rules. In either case, the *hamza* combined with the carrier is distributed back to the input.

```
parseHamza =  do  h <- oneof [hamza]
                  e <- inspectEnv
                  let combineWithCarrier = if envVerb e
                                 then parseVerbHamza h
                                 else parseArabHamza h
                  ;       do  lower [h] []
                              b <- combineWithCarrier
                              lower [] [b, b]
                   <|>
                          do  lower ["-", h] []
                              b <- combineWithCarrier
                              lower [] [b, "-", b]
                   <|>
                          do  b <- combineWithCarrier
                              lower [] [b]
                  parseNothing

parseVerbHamza :: [Char] -> Extend Env [Char] [Char]
parseVerbHamza h =
            do  i <- inspectIList
                case i of
                    x : y   -> returnIList ((h ++ x) : y)
                    _       -> returnIList [h]
                oneof [bound]
```

We provide the definition of `parseArabHamza` at the end of this chapter, for we believe it has never been published in such a complete and formal way. The algorithm essentially evaluates the position of the *hamza* in the word, and the context of vowels and consonants.

9.3.8 Encode Arabic ArabTEX ZDMG

This module relates the ArabTEX notation and the ZDMG phonetic transcription. The organization of the module is very similar to the previous one.

```
module Encode.Arabic.ArabTeX.ZDMG (ZDMG (..)) where

import Encode
import Encode.Mapper
import Encode.Extend
import PureFP.OrdMap

data ZDMG = ZDMG | Trans deriving (Enum, Show)

instance Encoding ZDMG where
  encode _ = error "'encode' is not implemented"
  decode _ = concat . parseFull decoderParsing .
             concat . parseLongest decoderMapping
```

Let us therefore only show how capitalization is implemented. The lexicon stores diacritized lowercase characters used as the standard phonetic transcription. Capitalization is possible, but *hamza* `'` and *ayn* `` ` `` are ‘transparent’ to it and let capitalize the following letter.

```
minor = define [
    ( "'",  [ 0x02BE ] ),    ( "`",  [ 0x02BF ] ) ]

sunny = define [ ( "t",      [ 0x0074 ] ),
                 ( "_t",     [ 0x0074, 0x0331 ] ),
                 ( "d",      [ 0x0064 ] ),
                 ( "_d",     [ 0x0064, 0x0331 ] ),
                 ( "r",      [ 0x0072 ] ),
                 ( "z",      [ 0x007A ] ),
                 ( "s",      [ 0x0073 ] ),
                 ( "^s",     [ 0x0073, 0x030C ] ),
                 ( ".s",     [ 0x0073, 0x0323 ] ),
                 ( ".d",     [ 0x0064, 0x0323 ] ),
                 ( ".t",     [ 0x0074, 0x0323 ] ),
                 ( ".z",     [ 0x007A, 0x0323 ] ),
                 ( "l",      [ 0x006C ] ),
                 ( "n",      [ 0x006E ] )          ]

parseSingleCons =
```

```
                do  c <- oneof [consonant, extra, invis]
                    x <- upperWith consControl
                                [c] [consonant, extra, invis]
                    resetEnv setCap False
                    parseSyllVowel c x
        <|>
                do  c <- oneof [minor]
                    x <- upper [c] [minor]
                    parseSyllVowel c x

consControl :: OrdMap m => [Char] -> [m [Char] [UPoint]]
                                  -> Environ -> [[UPoint]]
consControl x l e = if envCap e
                        then [ capFirst n | n <- noChange ]
                        else noChange
    where noChange = lookupList x l
          capFirst []     = []
          capFirst (x:xs) = (toEnum . flip (-) 0x0020 .
                                      fromEnum) x : xs
```

9.3.9 Discussion

Next to the original ArabTEX parser (Lagally, 1992, 2004), there is an implementation in Perl of the Encode Mapper and Encode Arabic modules (Smrž, 2003–2007) with which the interpreter is built as a multi-layer finite-state automaton. The method used there, however, does not achieve the elegance, clarity nor flexibility as the presented Haskell implementation. Lazy construction of the automaton and the power of functional combinator parsing is simply missing there.

The significance of the ArabTEX notation, devised with modifications also for languages other than Arabic, in lexicography, linguistics, and education is discussed in (Lagally, 1994).

Our motivation for developing this approach was the use of the notation in computational systems for Arabic language modeling, such as in ElixirFM. Further extensions of our work are expected, and inclusion of the programming library in various information processing systems is a possible next application.

This is the complete parser for determining the carrier of *hamza* from the context, according to the rules of Arabic orthography.

```
parseArabHamza :: [Char] -> Extend Env [Char] [Char]
parseArabHamza h =
                do  e <- inspectEnv
                    b <- prospectCarrier
                    let carryHamza = case envEarly e of

                            []          -> case b of
```

```
                        "'y"    ->  "'i"
                        "'i"    ->  "'i"
                        "'A"    ->  "'A"
                        _       ->  "'a"

        "i"  : _    ->              "'y"
        "_i" : _    ->              "'y"
        "e"  : _    ->              "'y"

        "I"  : _    -> caseofMultiI b
        "_I" : _    -> caseofMultiI b
        "E"  : _    -> caseofMultiI b
        "^I" : _    -> caseofMultiI b

        ["", "y"]   -> caseofMultiI b

        "u"  : _    -> caseofVowelU b
        "_u" : _    -> caseofVowelU b
        "o"  : _    -> caseofVowelU b

        "U"  : _    -> caseofMultiU b
        "_U" : _    -> caseofMultiU b
        "O"  : _    -> caseofMultiU b
        "^U" : _    -> caseofMultiU b

        "a"  : _    -> caseofVowelA b
        "_a" : _    -> caseofVowelA b

        "A"  : _    -> caseofMultiA b
        "^A" : _    -> caseofMultiA b

        ["", "'A"]  -> caseofMultiA b

        ""   : _    -> case b of

                        "'y"    ->  "'y"
                        "'w"    ->  "'w"
                        "'A"    ->  "'A"
                        "'a"    ->  "'a"
                        _       ->  "'|"

        _           -> error "Other context"

case carryHamza of

        "'A"    -> lower ["A"] []
        _       -> return []

return carryHamza
```

```
          where prospectCarrier = do  parseQuote
                                      b <- lookaheadCarrier
                                      lower [] ["\\\""]
                                      resetEnv setQuote False
                                      return b
                                  <|> lookaheadCarrier

                caseofMultiI b = case b of
                                          "'i"    ->  "'|"
                                          "'|"    ->  "'|"
                                          _       ->  "'y"

                caseofMultiU b = case b of
                                          "'i"    ->  "'|"
                                          "'|"    ->  "'|"
                                          "'y"    ->  "'y"
                                          _       ->  "'w"

                caseofMultiA b = case b of
                                          "'y"    ->  "'y"
                                          "'w"    ->  "'w"
                                          _       ->  "'|"

                caseofVowelU b = case b of
                                          "'y"    ->  "'y"
                                          _       ->  "'w"

                caseofVowelA b = case b of
                                          "'y"    ->  "'y"
                                          "'w"    ->  "'w"
                                          "'i"    ->  "'i"
                                          "'A"    ->  "'A"
                                          _       ->  "'a"

                lookaheadCarrier =

                          do  v <- oneof' '-' [multi, other] <|>
                                   oneof [multi, other]
                              let carryHamza = case v of

                                      "I"     ->  "'y"
                                      "_I"    ->  "'y"
                                      "^I"    ->  "'y"
                                      "E"     ->  "'y"

                                      "U"     ->  "'w"
                                      "_U"    ->  "'w"
                                      "^U"    ->  "'w"
                                      "O"     ->  "'w"
```

```
                    "A"     ->  "'A"

                    "aNA"   ->  "'N"

                    _       ->  "'a"

            lower [] [v]
            return carryHamza
<|>
        do  v <- oneof [vowel, nuuns] <|> return ""
            c <- oneof [sunny, moony, taaaa, invis, silent]
            let carryHamza = case v of

                    "i"     ->  "'y"
                    "iN"    ->  "'y"
                    "_i"    ->  "'y"
                    "e"     ->  "'y"

                    "u"     ->  "'w"
                    "uN"    ->  "'w"
                    "_u"    ->  "'w"
                    "o"     ->  "'w"

                    "a"     ->  "'a"
                    "aN"    ->  "'a"
                    "_a"    ->  "'a"

                    _       ->  "'|"

            case v of   ""      -> lower [] [c]
                        _       -> lower [] [v, c]

            return carryHamza
<|>
        do  v <- oneof [vowel, nuuns] <|> return ""
            let carryHamza = case v of

                    "i"     ->  "'i"
                    "iN"    ->  "'i"
                    "_i"    ->  "'i"
                    "e"     ->  "'i"

                    _       ->  "'|"

            case v of   ""      -> lower [] []
                        _       -> lower [] [v]

            return carryHamza
```

Thoughts are not thoughts any more
Merely a feeling in disguise . . .

Lucie, the nightly *Noc*

Conclusion

In this thesis, we developed the theory of Functional Arabic Morphology and designed ElixirFM as its high-level functional and interactive implementation written in Haskell.

Next to numerous theoretical points on the character of Arabic morphology and its relation to syntax, we proposed a model that represents the linguistic data in an abstract and extensible notation that encodes both *orthography* and *phonology*, and whose interpretation is customizable. We developed a domain-specific language in which the lexicon is stored and which allows easy manual editing as well as automatic verification of consistency. We believe that the modeling of both the *written* language and the *spoken* dialects can share the presented methodology.

ElixirFM and its lexicons are licensed under GNU General Public License and are available on `http://sourceforge.net/projects/elixir-fm/`. We likewise publish the implementations of Encode Arabic, MorphoTrees, and ArabTEX extensions.

The omissions and imperfections that we have committed are likely to be improved with time. We intend to integrate ElixirFM closely with MorphoTrees as well as with both levels of syntactic representation in the Prague Arabic Dependency Treebank.

It is time to return to the beginning.

Bibliography

Aho, Alfred V. and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM* 18(6):333–340.

Al-Sughaiyer, Imad A. and Ibrahim A. Al-Kharashi. 2004. Arabic Morphological Analysis Techniques: A Comprehensive Survey. *Journal of the American Society for Information Science and Technology* 55(3):189–213.

Badawi, Elsaid, Mike G. Carter, and Adrian Gully. 2004. *Modern Written Arabic: A Comprehensive Grammar*. Routledge.

Baerman, Matthew, Dunstan Brown, and Greville G. Corbett. 2006. *The Syntax-Morphology Interface. A Study of Syncretism*. Cambridge Studies in Linguistics. Cambridge University Press.

Bar-Haim, Roy, Khalil Sima'an, and Yoad Winter. 2005. Choosing an Optimal Architecture for Segmentation and POS-Tagging of Modern Hebrew. In *Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages*, pages 39–46. Ann Arbor, Michigan: Association for Computational Linguistics.

Beesley, Kenneth R. 1997, 1998. Romanization, Transcription and Transliteration. `http://www.xrce.xerox.com/competencies/content-analysis/arabic/info/romanization.html`.

Beesley, Kenneth R. 1998a. Arabic Morphology Using Only Finite-State Operations. In *COLING-ACL'98 Proceedings of the Workshop on Computational Approaches to Semitic languages*, pages 50–57.

Beesley, Kenneth R. 1998b. Consonant Spreading in Arabic Stems. In *Proceedings of the 17th international conference on Computational linguistics*, pages 117–123. Morristown, NJ, USA: Association for Computational Linguistics.

Beesley, Kenneth R. 2001. Finite-State Morphological Analysis and Generation of Arabic at Xerox Research: Status and Plans in 2001. In *EACL 2001 Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 1–8. Toulouse, France.

Beesley, Kenneth R. and Lauri Karttunen. 2003. *Finite State Morphology*. CSLI Studies in Computational Linguistics. Stanford, California: CSLI Publications.

Bird, Steven and Patrick Blackburn. 1991. A Logical Approach to Arabic Phonology. In *EACL*, pages 89–94.

Buckwalter, Tim. 2002. Buckwalter Arabic Morphological Analyzer Version 1.0. LDC catalog number LDC2002L49, ISBN 1-58563-257-0.

Buckwalter, Tim. 2004a. Buckwalter Arabic Morphological Analyzer Version 2.0. LDC catalog number LDC2004L02, ISBN 1-58563-324-0.

Buckwalter, Tim. 2004b. Issues in Arabic Orthography and Morphology Analysis. In *Proceedings of the COLING 2004 Workshop on Computational Approaches to Arabic Script-based Languages*, pages 31–34.

Cavalli-Sforza, Violetta, Abdelhadi Soudi, and Teruko Mitamura. 2000. Arabic Morphology Generation Using a Concatenative Strategy. In *Proceedings of NAACL 2000*, pages 86–93. Seattle.

Chakravarty, Manuel M. T. 1999. Lazy Lexing is Fast. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, vol. 1722 of *Lecture Notes in Computer Science*, pages 68–84. London, UK: Springer-Verlag.

Chalabi, Achraf. 2004. Sakhr Arabic Lexicon. In *NEMLAR International Conference on Arabic Language Resources and Tools*, pages 21–24. ELDA.

Charette, François. 2007. ArabXƎTEX: An ArabTEX-like interface for typesetting languages in Arabic script with XƎLATEX. Tech. Rep. 2007/05.

Crochemore, Maxime, Costas S. Iliopoulos, Yoan J. Pinzon, and James F. Reid. 2000. A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem. In *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*. Hunter Valley, Australia.

Cuřín, Jan, Martin Čmejrek, Jiří Havelka, Jan Hajič, Vladislav Kuboň, and Zdeněk Žabokrtský. 2004. Prague Czech-English Dependency Treebank 1.0. LDC catalog number LDC2004T25, ISBN 1-58563-321-6.

Dada, Ali. 2007. Implementation of the Arabic Numerals and their Syntax in GF. In *ACL 2007 Proceedings of the Workshop on Computational Approaches to Semitic Languages: Common Issues and Resources*, pages 9–16. Prague, Czech Republic: Association for Computational Linguistics.

Daumé III, Hal. 2002–2006. Yet Another Haskell Tutorial. http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf.

Debusmann, Ralph. 2006. *Extensible Dependency Grammar: A Modular Grammar Formalism Based On Multigraph Description*. Ph.D. thesis, Saarland University.

Debusmann, Ralph, Oana Postolache, and Maarika Traat. 2005. A Modular Account of Information Structure in Extensible Dependency Grammar. In *Proceedings of the CICLing 2005 Conference*. Mexico City/MEX: Springer.

Diab, Mona, Kadri Hacioglu, and Daniel Jurafsky. 2004. Automatic Tagging of Arabic Text: From Raw Text to Base Phrase Chunks. In *HLT-NAACL 2004: Short Papers*, pages 149–152. Association for Computational Linguistics.

Ditters, Everhard. 2001. A Formal Grammar for the Description of Sentence Structure in Modern Standard Arabic. In *EACL 2001 Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 31–37. Toulouse, France.

El Dada, Ali and Aarne Ranta. 2006. Open Source Arabic Grammars in Grammatical Framework. In *Proceedings of the Arabic Language Processing Conference (JETALA)*. Rabat, Morocco: IERA.

El-Sadany, Tarek A. and Mohamed A. Hashish. 1989. An Arabic morphological system. *IBM Systems Journal* 28(4):600–612.

El-Shishiny, Hisham. 1990. A Formal Description of Arabic Syntax in Definite Clause Grammar. In *Proceedings of the 13th Conference on Computational Linguistics*, pages 345–347. Association for Computational Linguistics.

Evans, Roger and Gerald Gazdar. 1996. DATR: A Language for Lexical Knowledge Representation. *Computational Linguistics* 22(2):167–216.

Finkel, Raphael and Gregory Stump. 2002. Generating Hebrew Verb Morphology by Default Inheritance Hierarchies. In *Proc. of the Workshop on Computational Approaches to Semitic Languages*, pages 9–18. Association for Computational Linguistics.

Fischer, Wolfdietrich. 2001. *A Grammar of Classical Arabic*. Yale Language Series. Yale University Press, third revised edn. Translated by Jonathan Rodgers.

Forsberg, Markus and Aarne Ranta. 2004. Functional Morphology. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 213–223. ACM Press.

Habash, Nizar. 2004. Large Scale Lexeme Based Arabic Morphological Generation. In *JEP-TALN 2004, Session Traitement Automatique de l'Arabe*. Fes, Morocco.

Habash, Nizar and Owen Rambow. 2005. Arabic Tokenization, Part-of-Speech Tagging and Morphological Disambiguation in One Fell Swoop. In *Proceedings of the 43rd Annual Meeting of the ACL*, pages 573–580. Ann Arbor, Michigan.

Habash, Nizar and Owen Rambow. 2006. MAGEAD: A Morphological Analyzer and Generator for the Arabic Dialects. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the ACL*, pages 681–688. Sydney, Australia.

Habash, Nizar, Owen Rambow, and George Kiraz. 2005. Morphological Analysis and Generation for Arabic Dialects. In *Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages*, pages 17–24. Ann Arbor, Michigan: Association for Computational Linguistics.

Hajič, Jan, Eva Hajičová, Petr Pajas, Jarmila Panevová, Petr Sgall, and Barbora Vidová-Hladká. 2001. Prague Dependency Treebank 1.0. LDC catalog number LDC2001T10, ISBN 1-58563-212-0.

Hajič, Jan, Eva Hajičová, Jarmila Panevová, Petr Sgall, Petr Pajas, Jan Štěpánek, Jiří Havelka, and Marie Mikulová. 2006. Prague Dependency Treebank 2.0. LDC catalog number LDC2006T01, ISBN 1-58563-370-4.

Hajič, Jan, Otakar Smrž, Tim Buckwalter, and Hubert Jin. 2005. Feature-Based Tagger of Approximations of Functional Arabic Morphology. In *Proceedings of the Fourth Workshop on Treebanks and Linguistic Theories (TLT 2005)*, pages 53–64. Barcelona, Spain.

Hajič, Jan, Otakar Smrž, Petr Zemánek, Petr Pajas, Jan Šnaidauf, Emanuel Beška, Jakub Kráčmar, and Kamila Hassanová. 2004a. Prague Arabic Dependency Treebank 1.0. LDC catalog number LDC2004T23, ISBN 1-58563-319-4.

Hajič, Jan, Otakar Smrž, Petr Zemánek, Jan Šnaidauf, and Emanuel Beška. 2004b. Prague Arabic Dependency Treebank: Development in Data and Tools. In *NEMLAR International Conference on Arabic Language Resources and Tools*, pages 110–117. ELDA.

Hajičová, Eva and Petr Sgall. 2003. Dependency Syntax in Functional Generative Description. In *Dependenz und Valenz – Dependency and Valency*, vol. I, pages 570–592. Walter de Gruyter.

Hinze, Ralf and Johan Jeuring. 2003a. Generic Haskell: Applications. In R. Backhouse and J. Gibbons, eds., *Generic Programming: Advanced Lectures*, vol. 2793 of *LNCS*, pages 57–97. Springer.

Hinze, Ralf and Johan Jeuring. 2003b. Generic Haskell: Practice and theory. In R. Backhouse and J. Gibbons, eds., *Generic Programming: Advanced Lectures*, vol. 2793 of *LNCS*, pages 1–56. Springer.

Hodges, Wilfrid. 2006. Two doors to open. In *Mathematical Problems from Applied Logic I: Logics for the XXIst century*, pages 277–316. New York: Springer.

Holes, Clive. 2004. *Modern Arabic: Structures, Functions, and Varieties*. Georgetown Classics in Arabic Language and Linguistics. Georgetown University Press.

Hudak, Paul. 2000. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press.

Huet, Gérard. 2002. The Zen Computational Linguistics Toolkit. ESSLLI Course Notes, FoLLI, the Association of Logic, Language and Information.

Huet, Gérard. 2003. Lexicon-directed Segmentation and Tagging of Sanskrit. In *XIIth World Sanskrit Conference*, pages 307–325. Helsinki, Finland.

Huet, Gérard. 2005. A Functional Toolkit for Morphological and Phonological Processing, Application to a Sanskrit Tagger. *Journal of Functional Programming* 15(4):573–614.

Humayoun, Muhammad. 2006. *Urdu Morphology, Orthography and Lexicon Extraction*. Master's thesis, Göteborg University & Chalmers University of Technology.

Hutton, Graham and Erik Meijer. 1996. Monadic Parser Combinators. Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham.

Hutton, Graham and Erik Meijer. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming* 8(4).

Jabri, Youssef. 2006. Typesetting Arabic and Farsi with the Arabi package. The Users Guide. Tech. Rep. 2006/12, École Nationale des Sciences Appliquées, Oujda, Morocco.

Jones, Mark P. 1997. First-class Polymorphism with Type Inference. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496. Paris, France.

Jones, Mark P. 2000. Type Classes with Functional Dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. London, UK: Springer. ISBN 3-540-67262-1.

Karttunen, Lauri. 2003. Computing with Realizational Morphology. In *CICLing Conference on Intelligent Text Processing and Computational Linguistics*, pages 205–216. Springer Verlag.

Kay, Martin. 1987. Nonconcatenative Finite-State Morphology. In *Proceedings of the Third Conference of the European Chapter of the ACL (EACL-87)*, pages 2–10. ACL, Copenhagen, Denmark.

Kay, Martin. 2004. Arabic Script-Based Languages Deserve to Be Studied Linguistically. In *COLING 2004 Computational Approaches to Arabic Script-based Languages*, page 42. Geneva, Switzerland.

Kiraz, George Anton. 2001. *Computational Nonlinear Morphology with Emphasis on Semitic Languages*. Studies in Natural Language Processing. Cambridge University Press.

Kogai, Dan. 2002–2006. Encode. Character encodings module in Perl, http://perldoc.perl.org/Encode.html.

Konz, Ned and Tye McQueen. 2000–2006. Algorithm::Diff. Programming module registered in the Comprehensive Perl Archive Network, http://search.cpan.org/dist/Algorithm-Diff/.

Kremers, Joost. 2003. *The Arabic Noun Phrase. A Minimalist Approach*. Ph.D. thesis, University of Nijmegen. LOT Dissertation Series 79.

Lagally, Klaus. 1992. ArabTEX: Typesetting Arabic with Vowels and Ligatures. In *EuroTEX 92*, page 20. Prague, Czechoslovakia.

Lagally, Klaus. 1994. Using TEX as a Tool in the Production of a Multi-Lingual Dictionary. Tech. Rep. 1994/15, Fakultät Informatik, Universität Stuttgart.

Lagally, Klaus. 2004. ArabTEX: Typesetting Arabic and Hebrew, User Manual Version 4.00. Tech. Rep. 2004/03, Fakultät Informatik, Universität Stuttgart.

Leijen, Daan and Erik Meijer. 2001. Parsec: A Practical Parser Library. http://research.microsoft.com/~emeijer/.

Ljunglöf, Peter. 2002. *Pure Functional Parsing. An Advanced Tutorial*. Licenciate thesis, Göteborg University & Chalmers University of Technology.

Maamouri, Mohamed and Ann Bies. 2004. Developing an Arabic Treebank: Methods, Guidelines, Procedures, and Tools. In *Proceedings of the COLING 2004 Workshop on Computational Approaches to Arabic Script-based Languages*, pages 2–9.

Manning, Christopher D. and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge: MIT Press.

Marlow, Simon. 2001. Happy: The Parser Generator for Haskell. http://www.haskell.org/happy/.

Marlow, Simon. 2003. Alex: A Lexical Analyser Generator for Haskell. http://www.haskell.org/alex/.

McCarthy, John and Alan Prince. 1990a. Foot and Word in Prosodic Morphology: The Arabic Broken Plural. *Natural Language and Linguistic Theory* 8:209–283.

McCarthy, John and Alan Prince. 1990b. Prosodic Morphology and Templatic Morphology. In M. Eid and J. McCarthy, eds., *Perspectives on Arabic Linguistics II: Papers from the Second Annual Symposium on Arabic Linguistics*, pages 1–54. Benjamins, Amsterdam.

McCarthy, John J. 1981. A Prosodic Theory of Nonconcatenative Morphology. *Linguistic Inquiry* 12:373–418.

Mikulová, Marie et al. 2006. A Manual for Tectogrammatical Layer Annotation of the Prague Dependency Treebank. Tech. rep., Charles University in Prague.

Nelken, Rani and Stuart M. Shieber. 2005. Arabic Diacritization Using Finite-State Transducers. In *Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages*, pages 79–86. Ann Arbor.

Othman, Eman, Khaled Shaalan, and Ahmed Rafea. 2003. A Chart Parser for Analyzing Modern Standard Arabic Sentence. In *Proceedings of the MT Summit IX Workshop on Machine Translation for Semitic Languages: Issues and Approaches*, pages 37–44.

Panevová, Jarmila. 1980. *Formy a funkce ve stavbě české věty [Forms and Functions in the Structure of the Czech Sentence]*. Academia.

Pierce, Benjamin C. 2002. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press. ISBN 0-262-16209-1.

Ramsay, Allan and Hanady Mansur. 2001. Arabic morphology: a categorial approach. In *EACL 2001 Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 17–22. Toulouse, France.

Roark, Brian and Richard Sproat. 2007. *Computational Approaches to Morphology and Syntax*. Oxford University Press. http://catarina.ai.uiuc.edu/L406_06/Readings/rs.pdf.

Sgall, Petr. 1967. *Generativní popis jazyka a česká deklinace [Generative Description of Language and Czech Declension]*. Academia.

Sgall, Petr, Eva Hajičová, and Jarmila Panevová. 1986. *The Meaning of the Sentence in Its Semantic and Pragmatic Aspects*. D. Reidel & Academia.

Sgall, Petr, Jarmila Panevová, and Eva Hajičová. 2004. Deep Syntactic Annotation: Tectogrammatical Representation and Beyond. In *HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, pages 32–38. Association for Computational Linguistics.

Smrž, Otakar. 2003–2007. Encode::Arabic. Programming module registered in the Comprehensive Perl Archive Network, http://search.cpan.org/dist/Encode-Arabic/.

Smrž, Otakar. 2007. ElixirFM — Implementation of Functional Arabic Morphology. In *ACL 2007 Proceedings of the Workshop on Computational Approaches to Semitic Languages: Common Issues and Resources*, pages 1–8. Prague, Czech Republic: ACL.

Smrž, Otakar and Petr Pajas. 2004. MorphoTrees of Arabic and Their Annotation in the TrEd Environment. In *NEMLAR International Conference on Arabic Language Resources and Tools*, pages 38–41. ELDA.

Smrž, Otakar, Petr Pajas, Zdeněk Žabokrtský, Jan Hajič, Jiří Mírovský, and Petr Němec. 2007. Learning to Use the Prague Arabic Dependency Treebank. In E. Benmamoun, ed., *Perspectives on Arabic Linguistics*, vol. XIX. John Benjamins.

Soudi, Abdelhadi, Violetta Cavalli-Sforza, and Abderrahim Jamari. 2001. A Computational Lexeme-Based Treatment of Arabic Morphology. In *EACL 2001 Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 155–162. Toulouse.

Spencer, Andrew. 2004. Generalized Paradigm Function Morphology. http://privatewww.essex.ac.uk/~spena/papers/GPFM.pdf.

Sproat, Richard. 2006. Lextools: a toolkit for finite-state linguistic analysis. AT&T Labs. http://www.research.att.com/~alb/lextools/.

Stump, Gregory T. 2001. *Inflectional Morphology. A Theory of Paradigm Structure*. Cambridge Studies in Linguistics. Cambridge University Press.

Swierstra, S. Doaitse. 2001. Combinator Parsers: From Toys to Tools. In G. Hutton, ed., *Electronic Notes in Theoretical Computer Science*, vol. 41. Elsevier Science Publishers.

Swierstra, S. Doaitse and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In J. Launchbury, E. Meijer, and T. Sheard, eds., *Advanced Functional Programming*, vol. 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag.

Van Roy, Peter and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming*. Cambridge: MIT Press.

Versteegh, Kees. 1997a. *Landmarks in Linguistic Thought III: The Arabic Linguistic Tradition*. Routledge.

Versteegh, Kees. 1997b. *The Arabic Language*. Edinburgh University Press.

Wadler, Philip. 1985. How to Replace Failure by a List of Successes. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, vol. 201 of *Lecture Notes in Computer Science*, pages 113–128. New York, NY, USA: Springer-Verlag. ISBN 3-387-15975-4.

Wadler, Philip. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, vol. 925 of *Lecture Notes in Computer Science*, pages 24–52. London, UK: Springer-Verlag. ISBN 3-540-59451-5.

Wadler, Philip. 1997. How to Declare an Imperative. *ACM Computing Surveys* 29(3):240–263.

Wadler, Philip. 2003. A Prettier Printer. In J. Gibbons and O. de Moor, eds., *The Fun of Programming*, Cornerstones of Computing, pages 223–243. Palgrave Macmillan.

Wadler, Philip and Stephen Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM.

Yaghi, Jim and Sane Yagi. 2004. Systematic Verb Stem Generation for Arabic. In *COLING 2004 Computational Approaches to Arabic Script-based Languages*, pages 23–30. Geneva, Switzerland.

Žabokrtský, Zdeněk. 2005. *Valency Lexicon of Czech Verbs*. Ph.D. thesis, Charles University in Prague.

Žabokrtský, Zdeněk and Otakar Smrž. 2003. Arabic Syntactic Trees: from Constituency to Dependency. In *EACL 2003 Conference Companion*, pages 183–186. Budapest, Hungary.

Linguistic Index